ANSI C

Egon Zakrajšek

26. oktober 1997

Kazalo

Predgovor **11**

1 Prvi koraki **13**

1.1 Kode in števila . . . 13
  1.1.1 Reprezentacija števil . . . 13
  1.1.2 Osembitne kode . . . 14
1.2 Hello, world . . . 15
  1.2.1 Oblika C programov . . . 17
  1.2.2 Prevajanje programov . . . 17
1.3 Menjalnik . . . 19
  1.3.1 Komentarji . . . 20
  1.3.2 Spremenljivke . . . 20
  1.3.3 Prirejanje in relacije . . . 21
  1.3.4 Deljenje . . . 22
  1.3.5 Izpisovanje . . . 22
  1.3.6 Večkratne vrednosti . . . 23
  1.3.7 Zanka while . . . 24
  1.3.8 Sestavljeni stavek . . . 25
  1.3.9 Zanka do - while . . . 25
  1.3.10 Zanka for . . . 25
  1.3.11 Operatorji s prirejanjem . . . 26
  1.3.12 Povečevanje in zmanjševanje . . . 26
  1.3.13 Pogojni stavki . . . 27
  1.3.14 Pogojni izrazi . . . 27
  1.3.15 Znaki in nizi . . . 28
  1.3.16 Parametri programa . . . 29
  1.3.17 Pretvarjanje nizov v število . . . 30
  1.3.18 Datoteke . . . 30
  1.3.19 Vrednost funkcije main . . . 31
  1.3.20 Realna števila . . . 31
1.4 Štetje . . . 33
  1.4.1 Makro definicije . . . 34
  1.4.2 Vrednost prirejanja . . . 35
  1.4.3 Štetje znakov in vrstic . . . 36
  1.4.4 Štetje besed . . . 37
  1.4.5 Program count . . . 37
1.5 Funkcije . . . 39
  1.5.1 Funkcijski prototipi . . . 40

1.5.2 Prenašanje parametrov 41
1.5.3 Najdaljša vrstica 42
1.6 Dolžina funkcij 43

2 Tipi, operatorji in izrazi 45
2.1 Konstante 45
2.1.1 Cela števila 45
2.1.2 Desetiški, osmiški in šestnajstiški zapis 45
2.1.3 Znaki 46
2.1.4 Realna števila 46
2.1.5 Konstantni izrazi 46
2.1.6 Konstantni nizi 47
2.1.7 Enumeracije 48
2.2 Definicije 48
2.2.1 Definicije spremenljivk 48
2.2.2 Začetne vrednosti spremenljivk 49
2.2.3 Statične in avtomatične spremenljivke 49
2.2.4 Kvalifikacija const 50
2.3 Operatorji 50
2.3.1 Aritmetični operatoji 50
2.3.2 Prioritete in asociativnost 51
2.3.3 Relacijski operatorji 51
2.3.4 Povečevanje in zmanjševanje 52
2.3.5 Bitni operatorji 53
2.3.6 Operatorji s prirejanjem 54
2.3.7 Pogojni izrazi 55
2.3.8 Pretvarjanje tipov 56
2.3.9 Slučajna števila 59
2.3.10 Prioritete operatorjev in njihova asociativnost 59

3 Kontrolni stavki 63
3.1 Stavek in blok 63
3.2 Pogojni stavek 63
3.2.1 Veriga pogojev 65
3.3 Stavek switch 65
3.4 Zanke 66
3.4.1 Zanka while 66
3.4.2 Zanka for 67
3.4.3 Boljša verzija funkcije myatoi 68
3.4.4 Shellova metoda sortiranja 68
3.4.5 Operator vejica 68
3.4.6 Zanka do 68
3.5 Stavki break, continue in goto 69

4 Funkcije in struktura programov 73
4.1 Funkcije 73
4.1.1 Program mygrep 73
4.1.2 Zgradba funkcij 74
4.1.3 Komuniciranje med funkcijami 74
4.1.4 Program lint 75

4.1.5 Več izvornih datotek 75
4.1.6 Funkcija myatof 75
4.1.7 Preprosto računalo 76
4.2 Eksterni objekti 77
4.2.1 Pomnilniški razredi 77
4.2.2 Kategorije 77
4.2.3 Računalo 78
4.2.4 Program na več izvornih datotekah 80
4.2.5 Doseg imen 80
4.2.6 Program calc 82
4.2.7 Statične interne spremenljivke 83
4.2.8 Register spremenljivke 83
4.2.9 Bločna struktura 83
4.2.10 Inicializacija 84
4.2.11 Rekurzija 85
4.3 Predprocesor 86
4.3.1 Direktiva #include 86
4.3.2 Direktiva #define 86
4.3.3 Direktiva #if 89

5 Kazalci in večkratne vrednosti 93
5.1 Kazalci 93
5.1.1 Adresni operator & 93
5.1.2 Operator dereferenciranja * 94
5.1.3 Kazalčni argumenti funkcij 95
5.1.4 Zlorabljanje kazalčnih argumentov 95
5.1.5 Funkcija getint 96
5.2 Kazalci in večkratne vrednosti 97
5.2.1 Med kazalcem in večkratno vrednostjo ni razlike? 99
5.2.2 Kazalec ni večkratna vrednost 100
5.2.3 Kazalci kot argumenti funkcij 101
5.2.4 Del večkratne vrednosti 102
5.3 Adresna aritmetika 102
5.3.1 Dodeljevanje pomnilnika 103
5.4 Manipulacije z nizi 104
5.4.1 Konstantni nizi 104
5.4.2 Funkcija mystrcpy 105
5.4.3 Funkcija mystrcmp 106
5.5 Večkratni kazalci; kazalci na kazalce 107
5.5.1 Sortiranje vrstic 107
5.6 Večdimenzionalne tabele 107
5.6.1 Funkciji day_of_year in month_day 107
5.6.2 Večdimenzionalna tabela daytab 108
5.6.3 Funkcija month_name 109
5.6.4 Dvodimenzionalna tabela proti kazalcu na kazalec 109
5.6.5 Tabela imen 110
5.7 Argumenti z ukazne vrstice 110
5.7.1 Program mygrep1 110
5.7.2 Program mygrep2 111
5.8 Kazalci na funkcije 111

5.9 Dodeljevanje večdimenzionalnih tabel 113
5.10 Komplicirane deklaracije 114

6 Strukture 117
6.1 Pojem strukture 117
6.1.1 Struktura fraction 117
6.2 Strukturne spremenljivke 118
6.2.1 Inicializacija struktur 118
6.2.2 Doseganje komponent 118
6.2.3 Gnezdenje struktur 119
6.2.4 Legalne operacije s strukturami 119
6.3 Prenašanje struktur v funkcije 119
6.3.1 Aritmetika z ulomki 120
6.3.2 Kazalci na strukture 120
6.3.3 Gneča na vrhu prioritetne lestvice 121
6.4 Večkratne strukture 122
6.5 Kazalci na strukture 123
6.5.1 Samoreferencirajoče strukture 124
6.5.2 Iskanje po tabelah 126
6.6 Kategorija typedef 128
6.7 Unije 129
6.8 Bitna polja 132

7 Branje in pisanje 135
7.1 Preprosto branje in pisanje 135
7.1.1 Funkcija getchar 135
7.1.2 Funkcija putchar 136
7.1.3 Glava stdio.h 136
7.1.4 Program lower 136
7.2 Funkcija printf 137
7.3 Variabilne liste argumentov 139
7.3.1 Funkcija minprintf 140
7.4 Funkcija scanf 141
7.4.1 Funkciji fscanf in sscanf 143
7.5 Druge datoteke 143
7.5.1 Struktura FILE in funkcija fopen 144
7.5.2 Funkciji getc in putc 145
7.5.3 Datoteke stdin, stdout in stderr 145
7.5.4 Funkciji getchar in putchar 145
7.5.5 Funkciji fscanf in fprintf 145
7.5.6 Program mycat 145
7.5.7 Funkcija ungetc 146
7.5.8 Funkciji fgets in fputs 146
7.6 Razne funkcije 147
7.6.1 Glava string.h 147
7.6.2 Glava stdlib.h 148
7.6.3 Glava math.h 149

8 Sistemske funkcije **151**
8.1 Datoteke 151
8.1.1 Funkciji read in write 152
8.1.2 Program mycp 152
8.1.3 Funkcija mygetchar 153
8.1.4 Funkcija open 153
8.1.5 Program mycp1 154
8.1.6 Funkcija close 154
8.1.7 Funkcija remove 154
8.1.8 Funkcija lseek 155
8.1.9 Funkcija fseek 155
8.2 Funkcija myfopen in podobno 156
8.3 Dodeljevanje pomnilnika 156
8.3.1 Program xmymalloc 158

A Programi **161**
A.1 money.c 162
A.2 money2.c 164
A.3 money3.c 165
A.4 money4.c 166
A.5 myecho.c 167
A.6 money5.c 168
A.7 money6.c 170
A.8 mywc.c 172
A.9 count.c 173
A.10 power.c 174
A.11 longest.c 175
A.12 binsearch.c 177
A.13 proto.h 178
A.14 count2.c 179
A.15 myatoi.c 180
A.16 shellsort.c 181
A.17 reverse.c 182
A.18 itoa.c 183
A.19 mygrep.c 184
A.20 myatof.c 186
A.21 xcalc.c 187
A.22 calc.c 188
A.23 qsort.c 191
A.24 proto4.h 192
A.25 fib.c 193
A.26 alloc.c 195
A.27 sortlines.c 196
A.28 dayofyear.c 199
A.29 mygrep1.c 201
A.30 mygrep2.c 202
A.31 sortlines2.c 204
A.32 matrix.c 207
A.33 dcl.c 208
A.34 fract.c 211

A.35 keyword1.c 214
A.36 keyword2.c 217
A.37 freq.c 220
A.38 macro.c 223
A.39 minprintf.c 226
A.40 xcalc1.c 228
A.41 mycat.c 229
A.42 mycat2.c 230
A.43 mycp.c 231
A.44 mygetchar.c 232
A.45 mygetchar2.c 233
A.46 mycp1.c 234
A.47 mystdio.h 236
A.48 myfopen.c 238
A.49 myfillbuf.c 239
A.50 myiob.c 240
A.51 xmymalloc.c 241

B Referenčni priročnik 245
B.1 Uvod 245
B.2 Leksični dogovori 245
  B.2.1 Leksični atomi 245
  B.2.2 Komentarji 246
  B.2.3 Imena 246
  B.2.4 Ključne besede 246
  B.2.5 Konstante 246
  B.2.6 Nizi 248
B.3 Sintaktične oznake 248
B.4 Pomen imen 248
  B.4.1 Pomnilniški razred 249
  B.4.2 Osnovni tipi 249
  B.4.3 Izpeljani tipi 250
  B.4.4 Kvalifikacije tipov 250
B.5 Objekti in *lvrednosti* 250
B.6 Pretvarjanja 250
  B.6.1 Celoštevična promocija 251
  B.6.2 Celoštevilčna pretvarjanja 251
  B.6.3 Cela in realna števila 251
  B.6.4 Realna števila 251
  B.6.5 Aritmetična pretvarjanja 251
  B.6.6 Kazalci in cela števila 252
  B.6.7 Void 252
  B.6.8 Kazalci na void 253
B.7 Izrazi 253
  B.7.1 Generiranje kazalcev 253
  B.7.2 Primarni izrazi 253
  B.7.3 Priponski izrazi 254
  B.7.4 Unarni operatorji 256
  B.7.5 Prisile 257
  B.7.6 Multiplikativni operatorji 258

B.7.7 Aditivna operatorja 258
B.7.8 Operatorja pomika 259
B.7.9 Relacijski operatorji 259
B.7.10 Operatorja enakosti 259
B.7.11 Bitni IN operator 260
B.7.12 Bitni ekskluzivni ALI operator 260
B.7.13 Bitni inkluzivni ALI operator 260
B.7.14 Logični IN operator 260
B.7.15 Logični ALI operator 261
B.7.16 Pogojni operator 261
B.7.17 Prirejanja 261
B.7.18 Operator vejica 262
B.7.19 Konstantni izrazi 262
B.8 Deklaracije 263
B.8.1 Določila pomnilniškega razreda 263
B.8.2 Določila tipa 264
B.8.3 Deklaracije struktur in unij 264
B.8.4 Enumeracije 267
B.8.5 Deklaratorji 268
B.8.6 Pomen deklaratorjev 268
B.8.7 Inicializacija 271
B.8.8 Imena tipov 273
B.8.9 Deklaracija typedef 274
B.8.10 Ekvivalenca tipov 274
B.9 Stavki 275
B.9.1 Označeni stavki 275
B.9.2 Izrazni stavek 275
B.9.3 Sestavljeni stavek 275
B.9.4 Izbirni stavek 276
B.9.5 Iteracijski stavki 276
B.9.6 Skočni stavek 277
B.10 Eksterne deklaracije 278
B.10.1 Definicije funkcij 278
B.10.2 Eksterne deklaracije 280
B.11 Doseg in vezava 280
B.11.1 Leksični doseg 280
B.11.2 Vezava 281
B.12 Predprocesiranje 281
B.12.1 Trigrafična zaporedja 282
B.12.2 Lepljenje vrstic 282
B.12.3 Definicije makrojev in njihovo nadomeščanje 282
B.12.4 Vključevanje datotek 284
B.12.5 Pogojno prevajanje 285
B.12.6 Kontrola oštevilčenja 286
B.12.7 Generiranje napak 286
B.12.8 Pragmatični komentarji 286
B.12.9 Prazna direktiva 286
B.12.10Vnaprej definirana imena 286
B.13 Povzetek slovnice 287
B.13.1 Slovnica 287

B.13.2 Angleško-slovenski slovar terminalnih simbolov 293

C Standardna knjižnica **295**
C.1 Vhod in izhod: stdio.h 295
C.1.1 Operacije na datotekah 296
C.1.2 Formatiran izpis 297
C.1.3 Formatirano branje 300
C.1.4 Funkcije za branje in pisanje znakov 302
C.1.5 Funkcije za direktno branje in pisanje 302
C.1.6 Funkcije za pozicioniranje datotek 303
C.1.7 Funkcije za napake 303
C.2 Testi znakovnih razredov: ctype.h 304
C.3 Funkcije za manipulacije nizov: string.h 304
C.4 Matematične funkcije: math.h 306
C.5 Druge porabne funkcije: stdlib.h 307
C.6 Diagnostika: assert.h 309
C.7 Variabilne liste argumentov: stdarg.h 310
C.8 Nelokalni skoki: setjmp.h 310
C.9 Signali: signal.h 311
C.10 Funkcije za koledar in uro: time.h 311
C.11 Omejitve definirane z izvedbo: limits.h in float.h 313

Literatura **315**

Stvarno kazalo **316**

Predgovor

Pričujoča knjiga opisuje jezik `C`, kot ga določa `ANSI` standard. `ANSI C` je kot jezik podoben prejšnji verziji jezika (Kernighan & Ritchie `C`), vendar so med njima pomembne razlike.

V poglavju Prvi koraki je razložen skoraj ves C, vsekakor pa dovolj, da ga bralec lahko začne uporabljati. Preostala poglavja, Tipi, operatorji in izrazi, Kontrolni stavki, Funkcije in struktura programov, Kazalci in večkratne vrednosti, Strukture in Branje in pisanje podrobneje obdelajo snov, navedeno v naslovu poglavja.

Poglavje Sistemske funkcije je davek operacijskemu sistemu UNIX in pove, kako lahko napišemo funkcije, ki jih uporabljajo standardne knjižnice. To je edino poglavje, ki je tesneje naslonjeno na operacijski sistem UNIX. Preostala poglavja se sklicujejo na UNIX samo v toliko, da je občasno pod čto rečeno, da pod Unixom glej, na primer, `man atoi`, za opis standardne funkcije `atoi`.

V nadaljevanju je obsežen dodatek Programi, kjer so navedeni teksti programov, ki jih opisujemo. Ti teksti so v posebnem dodatku zato, da kontinuiteta razlage ne bi trpela. Vrh vsega so ti programi na voljo kot anonimen FTP.

Dodatek Referenčni priročnik podaja kompakten povzetek celega jezika. V primeru dvoumnosti je Referenčni priročnik dokončni razsodnik.

Podoben dodatek je Standardna knjižnica, kjer so naštete vse standardne funkcije, ki jih mora podpirati vsaka izvedba jezika `ANSI C`.

Poglavje 1

Prvi koraki

`C` je programski jezik, tako kot Pascal, Fortran in tako dalje. `C` je postal slaven, ko je Dennis Ritchie napisal prvo verzijo operacijskega sistema `UNIX` v jeziku `C`. `C` je potemtakem jezik za pisanje operacijskih sistemov, ampak to ni res. V jeziku `C` lahko pišemo poljubne programe, ne samo operacijskih sistemov. Resnici na ljubo, danes je večina računalniških programov napisana v jeziku `C`.

`C` obstaja v različnih inačicah. To je predvsem `ANSI C`, s katerim se bomo mi ukvarjali. Pred njim je bil v modi `K&R C` (Kernighan & Ritchie `C`), ki je živ še dandanes. Obstaja še `C++`, ki pa ni nič drugega kot predprocesor za `C`, nekaj takega, kot je Structran predprocesor za Fortran.

1.1 Kode in števila

1.1.1 Reprezentacija števil

Vsak programski jezik ima v ozadju svoj model računalnika, na katerem bo program tekel. Če se dejanski računalnik preveč razlikuje od modela, je pač prevajanje programov, napisanih v tem programskem jeziku, v dejanske ukaze računalnika bolj komplicirano, malodane nemogoče. Model računalnika, ki stoji v ozadju jezika `C`, je zelo preprost. Osnovna predpostavka tega modela je pomnilnik računalnika. V modelu jezika `C` je pomnilnik veliko število zaporednih zlogov (byte), zlog 0, zlog 1, zlog 2, ... Zlog je osem bitov pomnilnika. V eni interpretaciji (`unsigned`, nepredznačena) hrani en zlog poljubno število med nič in 255, skupaj 256 (= 2^8) kombinacij. V drugi interpretaciji (`signed`, predznačena) lahko hrani en zlog števila med -128 in $+127$, torej spet 256 različnih števil. Pomen kombinacije kot je $(10101010)_2$ je torej odvisen od interpretacije. V nepredznačeni interpretaciji pomeni $(10101010)_2$ število $2^7 + 2^5 + 2^3 + 2^1 = 128 + 32 + 8 + 2 = 170$, v predznačeni interpretaciji (in dvojiškem komplementu) pa pomeni $(10101010)_2$ število $-(01010110)_2 = -(2^6 + 2^4 + 2^2 + 2^1) = -(64 + 16 + 4 + 2) = -86$. Dolžnost računalnika (programa) je, da se odloči za pravilno interpretacijo zloga kot je $(10101010)_2$.

Podobna je zgodba z večjimi števili. Dva zaporedna zloga lahko pomenita dve števili med nič in 255 (-128 in 127), lahko pa ta dva zloga interpretiramo kot 16 bitno število, spet na dva načina, kot nepredznačeno število ali kot predznačeno število, torej kot število med nič in $65,535$ ali pa kot število med $-32,768$ in $32,767$. Štiri zaporedne zloge lahko interpretiramo kot nepredznačeno število med nič in $2^{32} - 1$ ali pa kot predznačeno število med -2^{31} in $2^{31} - 1$.

Program je torej odgovoren za pravilno interpretacijo tako glede širine kot glede predznačenosti. Pri korektnem programiranju teče to samo od sebe, lahko pa namenoma ali po pomoti napišemo program, ki med tekom spreminja interpretacijo. Če program počne to namenoma, je to umazano programiranje. Če počne to po pomoti, je program pač napačen, izpisal bo nepravilne rezultate. Spreminjanju interpretacije se moramo torej skrbno izogibati.

V zloge pomnilnika pa lahko zapisujemo tudi znake kot so `'a'`, `'b'`, `'4'`, `' '` in tako naprej. Kako lahko računalnik zapiše kakšen znak v en zlog pomnilnika? Spomnimo se, da je en zlog pač osem bitov pomnilnika, torej moramo shranjevati znake kot števila. Katero število pripada posameznemu znaku? Tej asociaciji pravimo znakovni kod. Najbolj znan znakovni kod je `ASCII` (American Standard Code for Information Interchange) kod. To je v resnici sedembitni kod, torej lahko zakodiramo 128 različnih znakov. Prednost omejitve na 128 znakov je naslednja: vsako število med nič in 127 je shranjeno v zlogu enako v predznačeni kot v nepredznačeni interpretaciji. Slaba stran sedembitnega koda pa je, da ne preostane nič prostora za znake, ki jih v ZDA ne uporabljajo.

1.1.2 Osembitne kode

V zadnjem času so postale moderne osembitne kode, ki imajo še dodatnih 128 znakov. Ti dodatni znaki imajo kot števila vrednosti odvisne od predznačenosti: v nepredznačeni interpretaciji imajo dodatni znaki vrednosti med 128 in 255, v predznačeni interpretaciji pa vrednosti med -1 in -128. Če torej uporabljamo razširjeni `ASCII` kod, moramo načelno vedeti, ali so znaki interpretirani kot predznačena ali kot nepredznačena števila. V večini slučajev je razlikovanje nepomembno, včasih pa je od sile važno. Težava je v tem, da `ANSI C` ne predpisuje predznačenosti tipa `char` (en zlog), ampak prepušča odločitev sestavljalcem `C` prevajalnikov. Rezultat je seveda zmeda. Te zmede v ZDA ne občutijo, ker pač uporabljajo sedembitno kodo, kjer je razlikovanje res brez pomena. Če pa hočemo na primer uporabljati vse slovenske črke, moramo najti zanje prostor v obstoječi kodi. V sedembitni `ASCII` kodi prostora ni. V osembitni `ASCII` kodi prostor sicer je, a je bolj ali manj razprodan. Možnost, ki jo uporabljamo v Sloveniji, da šumniki nadomeste šest standardnih `ASCII` znakov, "ki jih nihče ne potrebuje", je kajpada nesprejemljivo mašilo.

Rešitev prinaša `ISO` (International Standard Organization) standard. `ISO` 8859 je poln osembitni kod, ki se v začetnem delu, $0 - 127$, ujema z `ASCII` kodom, kar osrečuje Amerikance. Seveda je preostalih 128 znakov premalo, da bi pokrili vse evropske jezike, da o azijskih jezikih ne govorimo. `ISO` standard 8859 obstaja v devetih variantah $8859-1$, $8859-2$, ..., $8859-9$. Vse te različne variante se, kot rečeno, ujemajo v prvih 128 znakih z `ASCII` kodom, v zadnjih 128 znakih pa so različne. Slovenske šumnike najdemo v kodi $8859-2$ (Latin 2 Code Set). To seveda pomeni, da moramo najti način, kako naj dodatne znake vnesemo (natipkamo), kako naj jih računalnik prikaže na ekranu in kako naj jih tiskalniki natiskajo, pa tudi, kako naj računalniku dopovemo, katero inačico `ISO` 8859 standarda uporabljamo. `ISO` 8859 standard sicer rešuje vsa ta vprašanja, vendar so računalniki, ki obvladajo popoln `ISO` standard, na žalost redki. Ker pa je to problem, ki tare vso Evropo, zgodba ni brezupna, saj bomo Slovenci dobili rešitev hkrati z drugimi Evropejci. Zaenkrat je najbolje, da se slovenskim črkam izogibamo kolikor je to mogoče.

1.2 Hello, world

Začnimo z jezikom `C`. Za vzorec si oglejmo program, napisan v jeziku `C`. To bo seveda slavni program `hello`, ki izpiše tekst `hello, world` na ekran. Program, ki je spodaj prikazan, je opremeljen s številkami vrstic. Razume se, da te številke *ne* spadajo k programu ampak so napisane samo zato, da bomo o programu lažje govorili.

```
1:  #include <stdio.h>
2:
3:  int
4:  main(void)
5:  {
6:      (void)printf("Hello, world\n");
7:      return 0;
8:  }
```

Že o tako kratkem programu je veliko povedati.

Vrstica 1 je direktiva, ki pove, da moramo vrstico 1 nadomestiti z vsebino standardne datoteke `<stdio.h>`, ki jo bo prevajalnik našel "na standardnem mestu"[1], kjerkoli to že je. Datoteki, ki jo tako vključimo v program, pravimo glava. Standardna glava `<stdio.h>` vsebuje vrsto definicij in deklaracij, ki jih potrebuje skoraj vsak program. Namesto da bi te definicije in deklaracije sami pisali, pač vključimo standardno glavo `<stdio.h>` v naš program. Podrobnosti o tem kasneje.

Program v jeziku `C` tvori vrsta definicij. Ena od njih mora biti definicija funkcije `main`, ki vrne količino tipa `int`, torej standardno (predznačeno) število. To število, vrednost funkcije `main`, interpretira operacijski sistem kot končno stanje (termination status) programa. Mehanizem, ki poskrbi za to, je izven naše kontrole.

O funkcijah bomo govorili kasneje. Zaenkrat samo privzemimo, da ima glava definicije funkcije `main` tako obliko, kot jo kažeta vrstici 3 in 4.

Vrstica 5 označuje začetek sestavljenega stavka, vrstica 8 pa njegov konec. O sestavljenih stavkih bomo govorili kasneje.

Vrstici 6 in 7 tvorita jedro funkcije `main`. Vrstica 7 pove kakšno vrednost ima funkcija `main`. Kot rečeno, je funkcija `main` tipa `int`, kar pomeni, da je njena vrednost tipa `int`. To vrednost posreduje operacijski sistem lupini (shell), ki lahko počne z njo, kar jo je volja. Običaj je, da programi (funkcije `main`) vrnejo vrednost nič, če je vse v redu in neničelno vrednost, če je program odkril kakšno napako. Zaenkrat nas ta del programiranja ne skrbi in bomo do nadaljnjega vsak program (vsako funkcijo `main`) končali z vrstico 7.

Vrstica 6 je pravo jedro programa. `C`, za razliko od drugih programskih jezikov, nima posebnih stavkov za opravljanje vhodno-izhodnih operacij (branje in pisanje). `C` doseže to s standardnimi funkcijami in `printf` je ena od njih. Deklarirana je v datoteki `<stdio.h>` kot

```
int printf(const char *fmt, ...)
```

---

[1] Pod `UNIX`-om je to imenik `/usr/include`

Funkcija `printf` izpiše na ekran to, kar pravijo njeni argumenti, v obliki, ki jo definira njen prvi argument, `fmt`. Vrednost funkcije, tipa `int`, je število znakov, ki jih je funkcija izpisala, ali vrednost `EOF`, če je šlo kaj narobe. Ker nas vrednost funkcije `printf` ne zanima, vrnjeno vrednost eksplicitno zavržemo s prisilo `(void)`.

Funkcija `printf` je ena od bolj kompliciranih standardnih funkcij, vendar jo potrebujemo že zelo zgodaj, še preden jo znamo pošteno opisati. Že njeni argumenti so komplicirani. Težavi se bomo za sedaj izognili in povedali samo, kako to funkcijo uporabljamo, da izpišemo konstanten, to je vedno isti, niz znakov. To seveda ni zelo zanimiv primer, a za začetek bo zadoščal. V teh okoliščinah ima funkcija `printf` en sam argument, niz znakov, ki jih hočemo izpisati, v narekovajih. Niz znakov v narekovajih s tehničnim izrazom imenujemo "niz".

Denimo, da vrstico 6 zamenjamo s preprostejšo varianto

```
(void)printf("Hello, world");
```

Če bi tak program prevedli in pognali, bi sicer še vedno izpisal na ekran tekst `Hello, world`, vendar bi kurzor ostal na koncu vrstice. Po eni strani nam to ugaja, ker lahko tekst sestavljamo po koščkih, na primer

```
(void)printf("Hello, ");
(void)printf("world");
```

Rezultat je isti kot zgoraj.

Tekst lahko izpisujemo po koščkih, po nekaj znakov naenkrat, ali pa vse hkrati. Ampak če je temu tako, potrebujemo mehanizem, s katerim bomo terminalu povedali, da je vrstica končana in da je nimamo namena po delih nadaljevati. To dosežemo s posebnim znakom za konec vrstice. Znak za konec vrstice je tudi eden od `ASCII` (ali `ISO`) znakov, vendar nekoliko neobičajen. Morda bi hoteli program napisati takole

```
(void)printf("Hello, world
");
```

vendar to ne doseže zaželenega učinka. Prevajalnik bo protestiral, da mora biti niz (v narekovajih) ves v eni vrsti. Tu se srečamo z dvoličnostjo pomena: znak za konec vrste smo stalno uporabljali, da smo ločevali vrstice programa med seboj. Sedaj pa potrebujemo znak za konec vrstice v drugačnem pomenu, kot znak, ki ga bo funkcija `printf` izpisala. Če pa napišemo

```
(void)printf("Hello, world\n");
```

standardnega znaka za konec vrstice (tipka `RETURN` ali `ENTER`) v nizu ni, tipka `ENTER` sledi podpičju, zato pa smo v niz dodali dva znaka, `'\n'`, nagibnico in črko `n`. Prevajalnik vidi ta dva zaporedna znaka in ju nadomesti z znakom za konec vrstice, ki ga sicer v niz ne smemo vtakniti. Torej, namesto da bi uporabili znak za novo vrsto, ta znak *opišemo* in prevajalnik ga bo vtaknil tja, kamor je treba.

Kaj še lahko povemo o našem programu? Vsak stavek, razen sestavljenega, zaključimo s podpičjem. V Pascalu podpičje ločuje stavke, v jeziku `C` pa se vsak stavek konča s podpičjem. Natančneje rečeno, konstrukcija brez podpičja je izraz, ko pa dodamo podpičje, postane stavek.

1.2.1 Oblika C programov

`C` program je formalno napisan v prostem formatu. To pomeni, da je program zaporedje simbolov, ki jih razdelimo na vrstice in ločimo s presledki, splošneje, z belimi znaki, kot nam pač ugaja. Med bele znake spadajo presledek, predelčnik, znak za novo vrsto in še kaj. Seveda obstajajo neformalna pravila o obliki programa. Nekatera od teh neformalnih pravil imajo globlji, ne samo estetski pomen.

Eno od takih neformalnih pravil pravi, naj se ime funkcije v definiciji funkcije, `main` v našem primeru, vedno začne v novi vrsti v prvi koloni. Razlog za to neformalno pravilo je naslednji. `UNIX` ponuja programerju vrsto koristnih orodij. Eno od njih je urejevalnik `vi`. Le ta nam omogoča, da poiščemo v tekstu poljubno besedo. Posebno imeniten primer iskanja je iskanje besede, ki se začne v začetku vrstice. Tudi druga orodja, na primer `grep`, poznajo to možnost. Deklaracijo funkcije v obsežnem programu je mnogo lažje najti, če se lahko zanesemo, da je program napisan tako, da se ime funkcije v definiciji vedno začne v prvi koloni.

Drugo neformalno pravilo zadeva uporabo velikih in malih črk. Formalno pravilo pravi, da so velike črke drugačni znaki kot male črke. Formalno lahko uporabljamo ene ali druge ali oboje. Neformalno pravilo pa dodaja, naj bodo imena `C` objektov (funkcij, spremenljivk, tipov, ...) pisana z malimi črkami, imena predprocesorjevih objektov pa z velikimi črkami.

1.2.2 Prevajanje programov

Vsak `C` program moramo zapisati na neko datoteko, kar pomeni, da moramo najti zanjo ime. `C` prevajalnik je uvedel naslednja, bolj ali manj formalna, pravila:

- datoteke z izvornimi `C` programi imajo imena, ki se končujejo na `.c`;
- datoteke z izvornimi programi v zbirniku imajo imena, ki se končujejo na `.s`;
- imena glav, ki jih `.c` datoteke vključujejo (`#include` direktive), imajo imena, ki se končujejo na `.h`;
- imena datotek, ki hranijo prevedene programe, se končujejo na `.o`;
- knjižnice `.o` datotek imajo imena, ki se končujejo na `.a`.

Pri tem velja pripomniti, da pod `UNIX`-om imena datotek niso strukturirana kot pod DOS-om (ime . podaljšek). Ime datoteke sicer lahko vsebujejo piko, ki pa ne pomeni nič posebnega.

Povezanemu programu, to je programu, zlepljenem iz `.o` in `.a` datotek, je ime `a.out`, če posebej ne zahtevamo drugače.

Ob tem na kratko povejmo, kako program prevedemo in kaj se godi med prevajanjem. Če je naš izvorni program shranjen v datoteki `hello.c` in če želimo, naj se zgrajeni program imenuje `hello`, `C` prevajalnik aktiviramo z ukazom

```
cc hello.c -o hello
```

Če napišemo samo

```
cc hello.c
```

se bo zgrajeni program imenoval `a.out`.

Prevajanje samo tvori vrsta faz.

Prva faza je vedno `C` predprocesor. `C` predprocesor o `C` jeziku pravzaprav zelo malo ve in to izkoriščamo, da včasih uporabljamo `C` predprocesor v kontekstu, ki s `C` programi nima nobene zveze. `C` predprocesor je približno makro procesor, karkoli to že pomeni. Njegova vloga je, da dani `C` program prepiše v pravi `C` program. Predprocesor v glavnem uboga predprocesorjeve direktive. To so vrstice, ki se začenjajo z višajem `#`. Pred višajem je lahko še kakšen bel znak. Eno takih direktiv smo že spoznali, `#include`, ki ima za posledico, da predprocesor zamenja `#include` vrstico z vsebino navedene datoteke, običajno neke `.h` datoteke. Kasneje bomo spoznali še druge predprocesorjeve direktive, kot so makro definicije, pogojno vključevanje teksta in podobno.

Rezultat predprocesorja prevzame pravi `C` prevajalnik, ki `C` program prevede v zbirni jezik računalnika, assembler. `C` prevajalnik sam je običajno sestavljen iz več podfaz, ki si podajajo predelan programski tekst druga drugi. Zadnja podfaza prevajalnika praviloma zgenerira `.s` datoteko, ki hrani program preveden v zbirnik.

Ta program, napisan v zbirniku, običajno vzame v roke še optimizator, ki skuša program optimizirati, to je skrajšati in pospešiti. Rezultat je spet program v zbirniku, upajmo da krajši in boljši.

Program, napisan v zbirniku, končno prevzame zbirnik, assembler, ki predela `.s` datoteko v `.o` datoteko. To je binarna, za ljudi nečitljiva datoteka. Zbirnik je v celi verigi prvi program, ki mora vedeti, kako je datoteka `.o` zgrajena.

Zadnja faza je povezovalnik, program `ld`. Ta pobere vse navedene `.o` in `.a` datoteke in jih združi v izvedljiv program, `a.out`, če s stikalom `-o` ne ukažemo drugače.

Zadnjo fazo, povezovalnik, lahko opustimo. Če aktiviramo prevajalnik z ukazom

```
cc -c hello.c
```

teče vse kot smo prej opisali, le povezovalna faza manjka, končni rezultat je datoteka `hello.o`. To je koristno, če je program sestavljen iz več kosov, vsak zapisan na svoji `.c` datoteki. Vsak kos zase lahko prevedemo in dobimo vrsto `.o` datotek. Program `cc` naslanja svoje odločitve na končnice kot so `.c`, na primer. Če vidi `.c` datoteko, jo prevede, če pa vidi `.o` datoteko, ve, da je že prevedena in `cc` poskrbi samo za povezovanje. Ko imamo torej vse `.o` datoteke, lahko ukažemo

```
cc prvi.o drugi.o tretji.o -o rezultat
```

in `cc` bo poskrbel, da bo povezovalnik pobral navedene `.o` datoteke in morda še kaj in zložil iz tega izvedljiv program, `rezultat`.

Ob tem opisu smo kajpada nekaj podrobnosti zamolčali. Kje je funkcija `printf`, ki jo naš program uporablja? "Vsakdo" ve, da tiči v knjižnici z imenom `libc.a`. To ve tudi program `cc` in ko pokliče na pomoč povezovalnik, mu navede vse `.o` datoteke, za katere ve, na koncu pa še ime `libc.a`, kjer bo povezovalnik našel manjkajoče funkcije, kot je `printf`.

Druga manjkajoča podrobnost je naslednje vprašanje. Tako imenovanega "glavnega programa" nismo napisali, napisali smo samo funkcijo z imenom `main`, ampak to je še zmerom samo funkcija, torej podprogram, ki ga mora nekdo poklicati, namreč "glavni program". Ta "glavni program" je za vse programe isti. Karkoli že počne, poklicati mora funkcijo z imenom `main`. Kasneje bomo videli, da ima funkcija `main` lahko tudi parametre, torej mora "glavni program" poskrbeti tudi za te parametre. Ta "glavni program" je običajno napisan v zbirniku in ga nihče nikoli ne vidi, kar ne pomeni, da ne obstaja. Program `cc` ve, kje je "glavni program" shranjen in kako se datoteka imenuje (nekaj.o). Potiho, ne da bi koga motil, `cc` naroči povezovalniku naj najprej vzame "glavni program", nato vse naše `.o` datoteke, in tako naprej. Ta zgodba implicira, naj nikar ne uporabljamo povezovalnika sami, saj ne primer ne vemo, kako se "glavni program" imenuje in kje je, program `cc` pa ima dovolj informacij, da lahko aktivira povezovalnik `ld` tako, kot se spodobi.

Naloga 1.1. Poženite program `hello`. Kaj se zgodi, če v program vpeljemo namerne (če je to sploh potrebno) napake, da na primer delčke programa izpustimo, po enega naenkrat.

1.3 Menjalnik

Naš naslednji program naj bo menjalnik denarja. Dani znesek naj razbije na bankovce (in kovance), recimo znesek 83456 SIT:

```
8 * 10000 = 80000
0 * 5000 = 0
3 * 1000 = 3000
0 * 500 = 0
2 * 200 = 400
0 * 100 = 0
1 * 50 = 50
0 * 20 = 0
0 * 10 = 0
1 * 5 = 5
0 * 2 = 0
1 * 1 = 1
```

Program A.1 je prikazan na strani 162.

Prve tri vrstice, namreč

```
#ifndef lint
static char *sccsid = "@(#)money.c  1.2";
#endif
```

ali nekaj podobnega, bomo videli zelo pogosto. Te vrstice za program v resnici ne pomenijo nič, v program so prišle kot posledica sistema za vzdrževanje tekstovnih datotek, SCCS. Vsi programi, navedeni v tem tekstu, so kajpada preizkušeni in da med prepisovanjem v knjigo ne bi prišlo do neljubih napak, je v končni tekst vključen program takšen, kot ga je prevajalnik videl. Takšne vrstice bomo pri branju preprosto ignorirali.

1.3.1 Komentarji

Sledeče vrstice so komentar, ki na kratko pove, kaj program počne. Vse znake med `/*` in prvim sledečim parom `*/` bo prevajalnik nadomestil z enim presledkom. To med drugim pomeni, da komentarji ne morejo biti gnezdeni (komentar znotraj komentarja). Komentar smemo napisati povsod, kjer smemo napisati kak bel znak. Komentarje praviloma pišemo v dveh oblikah. Kratke komentarje napišemo na koncu vrstice, ki jo opisujejo. Daljše komentarje uredimo v škatlo tako, da so zvezdice druga pod drugo.

Komentarje po možnosti pišemo v angleščini. Prej ali slej bo program prišel v roke ljudem, ki ne znajo slovensko, znajo pa zelo verjetno angleško. Priložnostni bralec programa se bo upravičeno kregal, če bodo komentarji napisani v slovenščini, ki je ne zna. Če pa ne razume angleško, je sam kriv, saj je angleščina danes najbolj razširjen jezik. Pred nekaj stoletji bi komentarje pisali v latinščini, ki je bila takrat jezik intelektualcev, danes pa je to angleščina, vsaj kar se računalništva tiče.

Spominjam se ameriškega profesorja, ki je pred leti obiskal ljubljansko univerzo in pripovedoval, kako je pred tem obiskal Portugalsko, kjer so mu ponudili vrsto zanimivih programov. Ampak vsi komentarji so bili portugalski ...

1.3.2 Spremenljivke

V (skoraj) vsakem programu uporabljamo spremenljivke. Spremenljivke so imena za koščke računalnikovega pomnilnika. Za vsako spremenljivko moramo navesti njen tip. To dosežemo z definicijami. Definicija je sestavljena iz tipa, ki mu slede imena spremenljivk, ki so s tem postale spremenljivke tega tipa, na primer

```
int     money;
```

Tip `int` pomeni, da ima spremenljivka celoštevilčno vrednost, za razliko od tipa `float`, ki pomeni, da ima spremenljivka realno vrednost, število v plavajoči vejici. Zaloga vrednosti takega tipa je odvisna od arhitekture računalnika in morda od prevajalnika. Tip `int` je običajno 16 bitno ali 32 bitno število, seveda predznačeno. Tudi `float` je pogosto 32 bitno število s kakimi sedmimi pomembnimi števkami in obsegom 10^{-38} do 10^{+38}. Detajli reprezentacije takih števil so precej komplicirani, a IEEE standard k sreči predpisuje, kako se morajo ti detajli vesti.

Formalno pravilo za izbiro imen pravi, da so imena sestavljena iz velikih in malih črk (angleške) abecede, števk in podčrtaja _, ki velja za črko. Neformalno pravilo pa dodaja, da imen ne začenjamo s podčrtajem, ker si taka imena pridržuje prevajalnik za svoje potrebe, in da uporabljamo načelno samo male črke, ker so imena, sestavljena iz velikih črk, "rezervirana" za predprocesorjeva imena. Imena, ki so sestavljena iz več "besed", pišemo tako, da "besede" ločimo s podčrtajem, ali pa vsako "besedo" začnemo z veliko začetnico, na primer

```
first_character, left_coordinate
FirstCharacter, LeftCoordinate
```

Seveda se držimo enega ali drugega dogovora.

Dolžina imen je formalno neomejena. Prevajalnik jamči, da bo upošteval vsaj prvih 31 znakov. Različna imena morajo biti v tem smislu različna. Prevajalnik na primer ne jamči, da bo opazil, da sta dve dolgi imeni, ki se ujemata v prvih 31 znakih, v resnici različni.

Imena se seveda ne smejo ujemati z rezerviranimi besedami kot so `if`, `while`, `int`, `float`, ...

`C` nudi še druge celoštevilčne tipe poleg `int`: `char`, `short` in `long`, ter druge realne tipe poleg `float`: `double` in `long double`. `ANSI` standard ne precizira, kako široki (koliko bitov) so to tipi, precizira le naslednje:

```
width(char) <= width(short) <= width(int) <= width(long)
```

in

```
width(float) <= width(double) <= width(long double)
```

kjer je `width(x)` širina tipa `x`. To pomeni, da ima `long` vsaj toliko bitov kot `int`, ki ima vsaj toliko bitov kot `short` in tako dalje. Tipične številke so

```
char         8
short       16
int         32
long        32

float       32
double      64
long double 96
```

Te širine običajno merimo z operatorjem `sizeof`. Po definiciji je `sizeof(char)` enak `1`, torej bi bile tipične širine, merjene s `sizeof` operatorjem

```
char         1
short        2
int          4
long         4

float        4
double       8
long double 12
```

Poleg številskih tipov obstajajo tudi večkratne vrednosti, strukture, unije, kazalci na te tipe in funkcije, ki vračajo te tipe. Vse to bomo še spoznali ob pravem času.

1.3.3 Prirejanje in relacije

Vrstica

```
money = 83456;
```

s katero se naš program v resnici začne, je prirejanje. Prirejanje je stavek oblike

```
spremenljivka = izraz;
```

Za prirejanje uporabljamo znak `=` in ne `:=` kot v Pascalu. To seveda pomeni rahlo komplikacijo, ko bo treba napisati relacijski operator enakosti. Le tega pišemo kot `==` (dva enačaja). Torej

```
money = 83456       /* money dobi vrednost 83456 */
money == 83456      /* vrednost money je enaka 83456,
                     * kar je morda res,
                     * ali pa tudi ne
                     */
```

Tu je popoln seznam relacijskih operatorjev:

`<`	manjši
`<=`	manjši ali enak
`>`	večji
`>=`	večji ali enak
`==`	enak
`!=`	neenak

1.3.4 Deljenje

Z našim programom priredimo torej spremenljivki `money` vrednost 83456, ki jo moramo razstaviti na bankovce in kovance. Postopek je preprost: začnemo pri najvišji denominaciji, `10000` SIT, izračunamo, koliko takih bankovcev moramo izplačati (`money / 10000`) in koliko še preostane, namreč `money % 10000`. Operator `/` je operator deljenja. Če sta oba operanda celi števili (`int`), in pri nas sta, pomeni to celoštevilčno deljenje (, kjer se ostanek izgubi). Celoštevilsko deljenje zaokroža proti ničli.

Ostanek pri deljenju z `10000` je znesek, ki ga moramo še izplačati z manjšimi bankovci. Operator `%` je operator ostanka:

```
money % 10000
```

je ostanek pri deljenju vrednosti `money` z `10000`. Ta ostanek hočemo shraniti nazaj v spremenljivko `money`, torej moramo reči

```
money = money % 10000;
```

1.3.5 Izpisovanje

Vrstica

```
(void)printf("%d * %d = %d\n", n, 10000, n * 10000);
```

izpiše tri števila, `n`, `10000`, in `n * 10000`, obenem z nekaterimi drugimi znaki. Kot vidimo, ima v tem primeru `printf` štiri argumente. Kot vedno je prvi argument formatni niz. Funkcija `printf` vse običajne znake v formatnem nizu izpiše kot doslej, posebnosti pa se začno z znakom `%`. Kombinacija `%d` v formatnem nizu pove, da moramo znaka `%d` nadomestiti v izpisu z vrednostjo naslednjega argumenta funkcije `printf`, pretvorjeno v decimalno število. Ta argument mora biti tipa `int`. Ker imamo v formatnem nizu trikrat napisano `%d`, morajo formatnemu nizu slediti tri celoštevilčne vrednosti, ki pa ni treba, da so spremenljivke, lahko so tudi izrazi, ki imajo celoštevilčno vrednost, na primer `n * 10000`.

Postopek moramo sedaj ponoviti z vsako manjšo denominacijo in to smo tudi storili, seveda na zelo neroden način. Zato bo naša prva naloga najti način, kako lahko v program vgradimo zanko, da ne bo treba vrstic ponavljati.

1.3.6 Večkratne vrednosti

Najprej potrebujemo seznam slovenskih bankovcev (in kovancev). Za začetek bomo pozabili na stotine. Imamo dvanajst denominacij, od bankovca za `10000` SIT navzdol do kovanca za `1` SIT. Te vrednosti je treba nekam shraniti. Če si izmislimo spremenljivke `v10000`[2], `v5000` in tako dalje, nismo nič na boljšem kot prej. Potrebujemo večkratno vrednost. Če definiramo

```
int banknote[12];
```

je `banknote` večkratna (dvanajstkratna) vrednost. Količina `banknote` je skupno ime za dvanajst vrednosti, namreč dvanajst različnih denominacij. Posamezne denominacije so `banknote[0]`, `banknote[1]`, ..., `banknote[11]`, kar nam pove dvoje: indeks večkratne vrednosti zapišemo v oglatih oklepajih za imenom večkratne vrednosti, pa tudi, da se indeksi štejejo od nič, ne od `1` kot v Fortranu. Na osnovi zgornje definicije imamo torej dvanajst vrednosti `banknote[i]`, kjer ima `i` vrednost med nič in enajst, vključno.

Seveda teh vrednosti še ni, samo prostor zanje je predviden v pomnilniku. En način, kako definiramo teh dvanajst vrednosti, je

```
banknote[0] = 10000;
banknote[1] = 5000;
```

in tako naprej. A v take namene pozna `C` inicializacijo spremenljivk. Namesto

```
int money;
money = 82345;
```

lahko definicijo in prirejanje združimo v celoto

```
int money = 82345;
```

Ta izboljšava ni kaj prida zanimiva, zanimiva pa postane, ko uporabimo zamisel na večkratni vrednosti. Namesto dvanajst prirejanj lahko napišemo definicijo večkratne vrednosti skupaj z inicializacijo:

```
int banknote[12] = { 10000, 5000, 1000, 500, 200, 100,
                     50, 20, 10, 5, 2, 1, };
```

torej kot del definicije za imenom spremenljivke napišemo še enačaj, ki mu sledi začetna vrednost spremenljivke, namreč seznam dvanajst vrednosti, ki tvorijo komponente večkratne vrednosti `banknote`.

V tej situaciji pa si lahko prihranimo štetje. Prevajalnik na osnovi take definicije ve, koliko komponent ima `banknote`, torej je odveč, da povemo, da jih je dvanajst in zadnjo definicijo poenostavimo v

---

[2] v10000 za vrednost 10000 SIT

```
int banknote[] = { 10000, 5000, 1000, 500, 200, 100,
                   50, 20, 10, 5, 2, 1, };
```

Prav tako ste gotovo opazili, da ima seznam v zavitih oklepajih na koncu (odvečno) vejico. Prevajalnik to namenoma dovoljuje, da je kar se da enostavno širiti seznam. To bomo začeli ceniti, ko bomo napisali komponente vsako v svojo vrsto. Tako spremenjen program A.2 je na strani 164.

1.3.7 Zanka while

Poleg definicije večkratne vrednosti `banknote` moramo še povedati, kako zgradimo zanko, ki bo iste stavke izvajala za vsako denominacijo. Mehanizmov za gradnjo zank je v jeziku `C` več. Najenostavnejši je zanka `while`, ki ima obliko

```
while (pogoj) stavek;
```

Pri tem je `pogoj` poljuben (aritmetičen) izraz, na primer `i < 12`. V drugih programskih jezikih bi temu rekli logičen izraz, ker je njegova vrednost `res` ali `ne res` (`true` ali `false`). V jeziku `C` logičnih spremenljivk ni, niti ni logičnih vrednosti (, ki bi bile drugačne od aritmetičnih). V jeziku `C` neničelna vrednost pomeni `true`, ničelna pa `false`. Tako

```
while (1) stavek;
```

pomeni neskončno zanko, saj je `pogoj 1` vedno izpolnjen. Ta koncept sega tako daleč, da lahko napišemo

```
i = j < k;
```

in spremenljivka `i` bo dobila neničelno vrednost `1` (`true`), če je `j` manjši od `k`, in ničelno vrednost, če to ni res.

Vsekakor, v zanki `while` se `pogoj` izračuna in če je izpolnjen (njegova vrednost `true`, torej neničelna), se `stavek` izvrši, zanka pa teče naprej tako, da ponovno izračuna `pogoj`.

V našem programu najprej postavimo števec (indeks) `i` na nič in nato

```
while (i < 12) {
...
}
```

V programu sicer ne napišemo `12` pač pa

```
sizeof(banknote) / sizeof(banknote[0]
```

ker smo se želeli izogniti štetju denominacij. Po kakšni logiki ima gornja konstrukcija vrednost `12`? `C` nam nudi operator `sizeof`. Tako je

```
sizeof(banknote)
```

velikost večkratne vrednosti `banknote`, merjena v zlogih,

```
sizeof(banknote[0])
```

pa ustrezna velikost ene (nulte) komponente. Ker je velikost večkratne vrednosti vedno velikost ene komponente pomnožene s številom komponent, je konstrukcija

```
sizeof(banknote) / sizeof(banknote[0])
```

število komponent v večkratni vrednosti `banknote`. Poudariti velja, da se ta vrednost izračunana med prevajanjem in da je (v našem primeru) to popolnoma enakovredno konstanti `12`.

Torej, komplikacije s `sizeof` na stran, zanko napišemo takole:

```
while (i < 12) stavek;
```

1.3.8 Sestavljeni stavek

Zelo poredko je en sam stavek v jedru zanke dovolj. Formalni zahtevi, da mora biti v jedru zanke en sam stavek, ugodimo tako, da uporabimo sestavljeni stavek. Če poljubno zaporedje stavkov vklenemo med zavite oklepaje, je cela konstrukcija sintaktično en sam stavek, ki ga lahko napišemo za pogojem v stavku `while`. To je univerzalno zdravilo. Pogosto bomo še srečali konstrukcije, ki terjajo po en sam stavek. Ta stavek je vedno lahko sestavljeni stavek, tako da omejitev na en stavek sploh ni nobena omejitev.

1.3.9 Zanka do - while

Stavek `while` pa ni edini način za konstrukcijo zank. Poleg stavka `while` poznamo še stavek `do`. Ta ima strukturo

```
do stavek while (pogoj);
```

in pomeni skoraj isto: izvajaj `stavek`, dokler je `pogoj` izpolnjen. Kje je razlika? Stavek `do` najprej izvede `stavek` in nato povpraša ali bi še. Stavek `while` pa najprej povpraša ali bi še in potem izvede `stavek`. V konstrukciji `while` se `stavek` morda ne bo izvedel niti enkrat, v konstrukciji `do` pa se bo zagotovo izvedel vsaj enkrat. Najbrž je odveč povedati, da stavek `do` uporabljamo bistveno manjkrat kot stavek `while`.

1.3.10 Zanka for

Tretja zančna konstrukcija je stavek `for`. Zamisel ja takale. Praviloma ima vsaka zanka najprej inicializacijo, nato `while` test, in končno korekturo, ki naj poskrbi, da ne bo zanka tekla v nedogled. Zanka v našem programu je tipičen primer. Inicializacija je stavek `i = 0;`, `while` test je `while (i < 12)` (ali karkoli že je namesto `12`), korektura pa stavek `i = i + 1;`. Take situacije so tako pogoste, da imamo zanje poseben stavek, stavek `for`, ki ima obliko

```
for (izraz1; izraz2; izraz3) stavek;
```

na primer

```
for (i = 0; i < 12; i = i + 1) { ... }
```

Praviloma je stavek `for` ekvivalanten konstrukciji

```
izraz1;
while (izraz2) {
    stavek;
    izraz3;
}
```

torej, izračunaj vrednost izraza `izraz1` (natanko enkrat), nato, dokler je vrednost izraza `izraz2` od nič različna (`true`), izvajaj `stavek` ter še izračunaj `izraz3`. Ekvivalenca `for` in `while` zanke se podre, če imamo v (sestavljenem) stavku `stavek` stavke `break` ali `continue`, a o tem kasneje.

1.3.11 Operatorji s prirejanjem

Še eno okrajšavo si oglejmo. Naj bo `op` neka operacija, na primer `+`. Pogosto naletimo na konstrukcijo

```
spremenljivka = spremenljivka op izraz
```

na primer

```
i = i + 1
money = money % banknote[i]
```

`C` nam ponuja operatorje s prirejanjem, namreč konstrukcijo

```
spremenljivka op= izraz
```

torej

```
i += 1
money %= banknote[i]
```

Te konstrukcije so popolnoma enakovredne originalnim z omejitvijo, da se leva stran izračuna samo enkrat. Ta omejitev je običajno brez posledic, na primer v zgornjem primeru. Le če ima leva stran stranske učinke, temu ni tako, na primer

```
x[i += 1] = x[i += 1] + 1
```

ni ekvivalentno konstrukciji

```
x[i += 1] += 1
```

1.3.12 Povečevanje in zmanjševanje

Obstaja še nadaljnja poenostavitev. Konstrukcijo `x += 1` lahko zamenjamo z `++x` ali `x++` in podobno za `x -= 1`. V čem je razlika med `++x` in `x++`? Obe konstrukciji povečata `x` za ena. Razlika je v vrednosti izraza `x++` oziroma `++x` samega. Izraz `x++` ima vrednost spremenljivke `x` pred povečanjem, izraz `++x` pa vrednost spremenljivke `x` po povečanju. Običajno temu pravimo takole: `++x` najprej poveča vrednost `x`, potem šele vzame vrednost spremenljivke `x`, pri `x++` pa obratno.

1.3.13 Pogojni stavki

Če pogledamo rezultate programa `money`, opazimo, da je nesmiselno izpisovati, da potrebujemo nič bankovcev po `500` SIT, zato hočemo te vrstice izpustiti iz izpisa. Z drugimi besedami, vrstice pišemo samo v primeru, da je število ustreznih bankovcev pozitivno.

To funkcionalnost nam omogoča stavek `if`, ki ima dve obliki

```
if (pogoj) stavek
```

pravi: če je pogoj `pogoj` izpolnjen, se izvrši stavek `stavek` (sicer pa nič). Oblika

```
if (pogoj) stavek1 else stavek2
```

pa pravi: če je pogoj `pogoj` izpolnjen, se izvrši stavek `stavek1` sicer pa stavek `stavek2`.

Vse te drobne spremenbe vodijo do nove verzije programa `money` in sicer do programa A.3 na strani 165.

1.3.14 Pogojni izrazi

Še eno lepotno spremembo napravimo. Rezultat izpišimo v obliki

```
3456 = 3 * 1000 + 2 * 200 + 1 * 50 + 1 * 5 + 1 * 1
```

iz katere je razviden začeten znesek in ves postopek z drobižem.

Večina sprememb je trivialna in se nanašajo na izpis brez znaka za konec vrste. Netrivialna sprememba je doseči, da na koncu vrstice nimamo odvečnega plusa, da rezultat ni videti takle:

```
3456 = 3 * 1000 + 2 * 200 + 1 * 50 + 1 * 5 + 1 * 1 +
```

V ta namen potrebujemo dvoje. Funkcija `printf` zna izpisovati tudi nekonstantne nize znakov, če v formatni niz (prvi parameter) napišemo `%s`. Določilo `%s` pomeni, da je naslednji parameter niz, na primer

```
printf("%s\n", "hello world");
```

Drugi pojem je pogojni izraz. To je konstrukcija oblike

```
pogoj ? izraz1 : izraz2
```

Njegova vrednost je `izraz1`, če je `pogoj` izpolnjen, in `izraz2`, če ni. Na primer

```
i > 0 ? +i : -i
```

je absolutna vrednost spremenljivke `i`. Ta zamisel deluje tudi na nizih in

```
money > 0 ? " + " : "\n"
```

pomeni niz `"␣+␣"`, če je `money` pozitiven in niz `"\n"`, če ni. Tak pogojen niz moramo izpisati na koncu vsake denominacije. Tako dobimo verzijo A.4 na strani 166.

1.3.15 Znaki in nizi

Naslednji ugovor k našemu programu je konstantna vrednost, ki jo razstavljamo na bankovce. Če hočemo razstaviti na bankovce kak drug znesek, ne `83456` SIT, moramo popraviti program, to pa ni lepo.

Od kod naj program `money` dobi znesek, ki ga mora razstaviti? Recimo, da program poženemo takole

```
money 65432
```

z namenom, da je `65432` znesek, ki ga hočemo razstaviti. Seveda mora biti to drugačna verzija programa `money`.

Da bi lahko o tem razpravljali, si moramo najprej ogledati natančneje, kaj so znaki in kaj so nizi znakov.

Količine tipa `char` so znaki. Ne vemo, ali so predznačeni ali ne, a to ni pomembno, dokler ostanemo znotraj 128 `ASCII` znakov. Vrednosti tipa `char` so na primer

`65`	znak A v ASCII kodi
`'A'`	tako znak A običajno pišemo
`66`	znak B v ASCII kodi
`'B'`	tako znak B običajno pišemo
`32`	presledek v ASCII kodi
`' '`	tako presledek običajno pišemo
`16`	znak za novo vrsto v ASCII kodi
`'\n'`	tako znak za novo vrsto običajno pišemo
`9`	predelčnik (TAB) v ASCII kodi
`'\t'`	tako predelčnik običajno pišemo
`0`	prazen znak
`'\0'`	prazen znak, kot ga običajno pišemo

Uporabljamo lahko tudi večkratne vrednosti, na primer

```
char t[4] = {'a', 'b', 'c', '\0'};
```

To so štirje znaki, imenovani `t[0]`, `t[1]`, `t[2]` in `t[3]`, kjer je `t[2] == 'c'` res (true). Tako pisanje nizov z naštevanjem znakov je nerodno. Lepše je

```
char t[4] = "abc";
```

To pomeni isto kot zgoraj, namreč znak `t[3]` je še vedno `'\0'`. Zakaj? Kadarkoli napišemo niz kot je

```
"hello, world"
```

morajo programi, ki tak niz znakov uporabljajo, znati najti dolžino niza. V Pascalu so nizi shranjeni tako, da je na začetku zapisana dolžina niza. V jeziku `C` temu ni tako, vsi programi pa vedo, da je vsak niz vedno shranjen kot večkraten znak, pri čemer je zadnji znak `'\0'`. To seveda pomeni, da prazen znak ne more biti na sredini niza, recimo takole

```
"abcde\0xyzuv"
```

Vsak program bo videl v tem nizu samo 5 znakov plus prazen znak na koncu.

Definicija

```
char t[4] = "abc";
```

je enakovredna definiciji

```
char t[] = "abc";
```

saj prevajalnik zna sam prešteti število znakov v nizu $(3 + 1)$.

Namesto definicije

```
char t[] = "abc";
```

lahko pišemo tudi

```
char *s = "abc";
```

Količina `t` je trikratna (v resnici štirikratna) vrednost, količina `s` je kazalec na znak in ta kazalec kaže na prvi znak v nizu "abc". Vsaj v tem kontekstu med obema definicijama ni razlike. Če pa definiramo

```
char t[10], *s;
```

je `t` desetkratna vrednost, v kateri zaenkrat še ni nobenega znaka, samo prostor zanje je predviden. Količina `s` je kazalec na znak, ki zaenkrat še ne ve, kam naj kaže. Če napišemo

```
s = "hello";
```

`s` sedaj kaže na prvi znak niza `"hello"`, torej na znak `'h'`. Po drugi strani, če napišemo

```
for (i = 0; i < 6; ++i)
    t[i] = s[i];            /* da, natanko tako */
```

pomenita `s` in `t` enak niz znakov. Ko smo se enkrat prebili čez definicijo, lahko uporabljamo večkratno vrednost kot večkratno vrednost ali pa kot kazalec, kazalec lahko uporabljamo kot kazalec ali kot večkratno vrednost.

Seveda obstaja razlika med kazalcem in večkratno vrednostjo, a o tej razliki bomo govorili kasneje, glej 5.2.

1.3.16 Parametri programa

Mehanizem, ki požene naš program[3], razbije vrstico, ki jo natipkamo, na "besede". Pri tem je "beseda" zaporedje znakov med belimi znaki. Če torej natipkamo

```
money 65432
```

smo natipkali dve besedi, `money` in `65432`. Ti dve besedi sta programu `money` dostopni, če ju želi pogledati. V takem primeru definiramo funkcijo `main` drugače kot doslej

---

[3] pod UNIX-om je to lupina, shell

```
int
main(int argc, char *argv[])
```

torej povemo, da ima funkcija `main` dva argumenta, število `argc` in `argv`, ki je večkraten kazalec na znak. Količina `argv[0]` je tedaj kazalec na začetek prvega (nultega) niza, `argv[1]` pa kazalec na začetek drugega niza, `argc` pa pove, koliko `argv`-jev je v večkratni vrednosti. Morda ste že uganili, da je `argv[0]` kazalec na ime, s katerim smo program zagnali, `argv[1]` pa kazalec na niz znakov, ki sledi imenu programa, če tak niz seveda obstaja (`argc >= 2`). Če torej napišemo program z imenom `myecho`, glej program A.5 na strani 167 in ga poženemo takole

```
myecho 125 + 345
```

bomo dobili

```
argv[0] = "myecho"
argv[1] = "125"
argv[2] = "+"
argv[3] = "345"
```

Ali ste opazili, kako skrbno je dvojnim narekovajem odvzet pomen z nagibnico? Če tega ne bi storili, bi se formatni niz končal za enačajem, ostanek bi bil pa nerazumljiva solata.

1.3.17 Pretvarjanje nizov v število

Sedaj vemo, kako priti do niza `"65432"`, če poženemo program z ukazom

```
money 65432
```

Težava je še v tem, da potrebujemo število 65432 in ne niza `"65432"`. Najti moramo način, kako niz znakov, ki ponazarja število, res prevedemo v število. V standardni glavi `<stdlib.h>` je deklaracija funkcije `atoi`[4], ki ima za argument kazalec na začetni znak niza. Znaki v tem nizu morajo tvoriti celo število in vrednost funkcije `atoi` je ravno to število, torej

```
atoi("65432")
```

je ravno 65432. Tako dobimo takle program A.6 na strani 168.

1.3.18 Datoteke

V tej verziji je še nekaj novosti, funkcija `fprintf`, ki je posplošitev funkcije `printf`, posplošitev v smislu, da lahko povemo, kam naj funkcija piše. Prvi argument funkcije `fprintf` je kazalec na nekaj, čemur pravimo `FILE`, prvi parameter funkcije `fprintf` je torej tipa `FILE*`. Kako take objekte definiramo in kako jih napravimo uporabne, nas ta hip ne zanima. Ta hip zadošča, da vemo, da datoteka `<stdio.h>`, ki smo jo vključili (`#include`) v naš program, vsebuje deklaracije treh takih kazalcev, namreč `stdin`, `stdout` in `stderr`. Funkcija `printf` ni nič drugega kot funkcija `fprintf` s prvim argumentom `stdout`, torej stavek

---

[4] Pod `UNIX`-om glej `man atoi`

```
(void)printf("%d = ", money);
```

ni nič drugega kot

```
(void)fprintf(stdout, "%d = ", money);
```

Objektom tipa `FILE` (ali `FILE*`) bomo rekli datoteke. Funkcija `printf` piše torej na datoteko `stdout`, funkcija `fprintf` pa na katerokoli datoteko. Datoteka `stdin` je namenjena za branje. V običajnih razmerah je `stdin` ime tipkovnice našega terminala, `stdout` in `stderr` pa dve imeni za ekran našega terminala. Običajno je torej vseeno ali pišemo na `stdout` ali pa na `stderr`. Pogosto pa z lupino poskrbimo, da sistem pošlje datoteko `stdout` kam drugam, recimo po "pipi" v `stdin` drugega programa. V takem primeru je pomembno, da samo normalne rezultate pišemo na `stdout`, opise napak pa na `stderr`.

1.3.19 Vrednost funkcije main

Program `money5` pravtako skrbno preveri, če smo program res pognali s še enim argumentom, z zneskom. Prav tako znesek ne sme biti enak nič ali celo negativen. (Kaj je narobe z zneskom nič?) V vsakem primeru, če je program napačno poklican, izpišemo na `stderr` primerno sporočilo, ki ponavadi vsebuje ime programa, namreč `argv[0]`.

V primeru napake funkcija `main` vrne neničelno število, `1` v našem primeru. Kakšno neničelno vrednost vrnemo v primeru napake je precej vseeno, pomembno pa je, da je vrnjena vrednost nič, če program ni odkril kakšne napake. Stavek

```
return 1;
```

na sredi funkcije `main` prekine izvajanje funkcije `main` in kdorkoli je funkcijo `main` poklical, bo videl vrnjeno vrednost. Vrednost, ki jo funkcija `main` vrne, najde pot v lupino, ki lahko to vrednost upošteva, če želi. V Kornovi lupini lahko na primer to vrednost izpišemo z ukazom

```
print $?
```

Seveda vsak ukaz na novo postavi `$?`, tako da moramo ukaz `print $?` uporabiti takoj po ukazu, katerega vrednost nas zanima.

1.3.20 Realna števila

Ostane nam še ena pomankljivost: program ne ve, da imamo v Sloveniji poleg tolarjev še stotine. To izkoristimo, da si ogledamo, kako ravnamo z realnimi števili.

Najprej imamo tri širine realnih števil: `float`, `double` in `long double`. Formalno so lahko vsi trije tipi enaki, mišljeno pa je, da je `double` natančnejši od `float`, `long double` pa natančnejši od `double`. Tip `long double` je pridobitev `ANSI` standarda. Včasih se prevajalniki pretvarjajo, da je `long double` isto kot `double`. Nasploh je tip `double` najbolj standarden. Tip `float` bi uporabljali le, da prihranimo nekaj pomnilnika, recimo 4 zloge na realno število, kar je absurdno, razen v primeru, da imamo miljonkratno vrednost. Teoretično utegne biti aritmetika s tipom `float` hitrejša od aritmetike s tipom `double`, verjetno pa to ni.

Na strani 170 vidimo program A.7, ki računa z realnimi števili, natančneje rečeno, s tipom `double`.

Kaj je novega?

Najprej smo vključili datoteko `math.h`, ki vsebuje deklaracije vseh "matematičnih" funkcij.

Količini `money` in `banknote` sta sedaj tipa `double` namesto `int`. V seznam denominacij smo dodali še vrednost 0.50 tolarja, to je 50 stotinov. To nam pove, da uporabljamo decimalno piko, ne decimalne vejice, in cela števila lahko napišemo tam, kjer prevajalnik pričakuje realna števila.

Znesek moramo spet dekodirati, le da namesto funkcije `atoi` uporabimo funkcijo `atof`[5], ki vrne realno (`double`) vrednost.

Znesek moramo zaokrožiti na najmanjšo denominacijo, to je 50 stotinov. Najmanjšo denominacijo bomo sicer tudi še potrebovali, zato smo definirali in inicializirali količino `small`, najmanjsi kovanec. Dobimo ga kot zadnjo komponento vrednosti `banknote`.

Znesek `money` kot poprej izpišemo, le da moramo funkciji `printf` povedati, da je količina, ki jo izpisujemo, realna in da hočemo videti dve decimalki. Formatno določilo `%f` pomeni izpis realnega števila, določilo `%.2f` pa pove, da hočemo videti dve mesti za decimalno piko.

Ko je enkrat znesek `money` izpisan, ne prej, `money` zaokrožimo tako, da prištejemo pol vrednosti najmanjšega kovanca.

Število bankovcev izračunamo kot prej. Kvocient je seveda realno število, prirejanje celoštevilčni spremenljivki `n` pa mu poreže decimalke.

Od zneska `money` moramo odšteti bankovce, ki smo jih že izplačali. To lahko storimo z odštevanjem

```
money -= n * banknote[i];
```

ali pa uporabimo funkcijo `fmod`[6], ki ima isti učinek, le kvocienta `n` zanjo ne potrebujemo.

Končno, pri zadnjem izpisu vrednosti `money` ni treba, da je le ta že nič, dovolj je, da je manjša od najmanjšega kovanca, `small`.

Pri prevajanju programa, natančneje pri povezovanju, moramo povedati, kje bo povezovalnik našel potrebne funkcije. Tega vprašanja doslej nismo načenjali, ker so bile vse potrebne funkcije v knjižnici `libc.a`, `cc` program pa ve, da je knjižnico `libc.a` vedno treba na koncu preiskati, če ne vsebuje morda kakšne manjkajoče funkcije. Funkcije `fmod` pa v knjižnici `libc.a` ni, ta funkcija je, skupaj z drugimi "matematičnimi" funkcijami, shranjena v knjižnici `libm.a`, zato moramo `cc` programu povedati, da bo povezovalniku povedal, da je knjižnica `libm.a` tudi izvor manjkajočih funkcij. Program `money6` napravimo torej takole:

```
cc -o money6 money6.c -lm
```

Dodatek `-lm` pove, naj povezovalnik preišče še knjižnico `libm.a`, kje jo bo našel, pa `cc` oziroma `ld` že ve.

---

[5] Pod UNIX-om glej `man atof`

[6] Pod UNIX-om glej `man fmod`

Naloga 1.2. Zadnja verzija programa `money`, `money6`, se čudno obnaša. Ukaz `money6 0.20` natiska `0.20 =`, in `money6 0.0020` izpiše `0.00 =`. Zakaj? Kako popraviti?

Naloga 1.3. Program `money5` popravite, da ne bo izpisoval `1 * 500` ampak samo `500`, kadar imamo opravka z enim samim bankovcem ali kovancem.

1.4 Štetje

Za pisanje (na `stdout` in `stderr`) smo uporabljali doslej funkcijo `printf` oziroma `fprintf`. Lahko pa uporabljamo tudi bolj preprosto funkcijo, `putchar`, in njeno sestrično, `getchar`. Funkcija `putchar` izpiše znak, ki ga podamo kot argument, na datoteko `stdout`, funkcija `getchar` (brez parametrov) pa prebere naslednji znak z datoteke `stdin` in ga vrne kot celoštevilčno vrednost (`int`).

Za vzorec napišimo program, ki prepiše vse znake z datoteke `stdin` na datoteko `stdout`. Program mora biti videti takle:

```
read a character
while (character is not end_of_file character)
    write out character just read
    read a character
```

Prepisati ta pseudo program v `C` sploh ni težko:

```
#include <stdio.h>

/* copy stdin na stdout; first version */

int
main(void)
{
    int     c;

    c = getchar();
    while (c != EOF) {
        (void)putchar(c);
        c = getchar();
    }

    return 0;
}
```

Funkcija `read a character` je kajpada `getchar`, `write out character` pa `putchar`.

Funkcija `getchar` je brez parametrov, oklepaje moramo pa kljub temu pisati, to je

```
getchar()
```

Vrednost funkcije `getchar` je tipa `int`, ne `char`. Razlog je v tem, da mora biti `getchar` sposobna vrniti katerikoli znak, ali pa povedati, da še enega znaka ni.

Tip `char` je preozek, da bi vanj lahko zapisali (ob različnih časih) vse vrednosti tipa `char` in še vrednost `EOF`. Zato je `getchar` deklarirana kot funkcija, ki vrne vrednost tipa `int`, in takega tipa mora biti tudi spremenljivka `c`, kamor to vrednost shranimo.

Funkcija `putchar` tudi vrne vrednost, in sicer znak, ki ga je ravnokar izpisala, če ji je to uspelo, in `EOF`, če iz tega ali onega razloga znaka ni mogla izpisati. V našem programu nas to ne zanima, zato vrednost funkcije `putchar` zavržemo s prisilo `(void)`.

1.4.1 Makro definicije

Kaj je `EOF`? To je neko število, recimo `-1`, ali kaj podobnega. Konstanta `EOF` je definirana v datoteki `<stdio.h>` s predprocesorjevo direktivo. `C` predprocesor prepiše izvorni program in pri tem nadomesti reference na `EOF` z ustrezno vrednostjo, karkoli že je. Taki konstrukciji kot je `EOF` pravimo makro. Programerji, ki so celo življenje programirali v Pascalu in se ne morejo privaditi na zavite oklepaje, lahko pišejo na primer programe takole:

```
#include <stdio.h>

#define BEGIN   {
#define END     }

int
main(void)
BEGIN
    int     c;

    c = getchar();
    while (c != EOF) BEGIN
        (void)putchar(c);
        c = getchar();
    END
END
```

Takih ekscesov iz utemeljenih razlogov nikakor ne priporočamo, čeprav ta trenutek še ne moremo povedati zakaj ne. Bolj smiselno je definirati "čarobne" konstante kot makro definicije, recimo

```
#define MAXLINE     1000

char line[MAXLINE];
....
```

Seveda pa tudi tu lahko pretiravamo, tako da je dejansko vrednost take makro definicije človeku skoraj nemogoče rekonstruirati. Nezmernost povsod škoduje, tudi pri programiranju.

Makro (brez parametrov) definiramo tako, da na začetku vrstice napišemo

```
#define
```

ki mu sledi ime makroja, ki ga definiramo. Imenu makroja sledi nekaj belih znakov, preostanek do konca vrstice pa je vsebina makroja. Predprocesor se ne briga za pomen vsebine. Ko vidi isto ime referencirano v tekstu programa, ime makroja zamenja z njegovo vsebino in prepusti prevajalniku, da ugotovi, ali je s tem nastala smiselna reč ali ne.

Kakšen smisel ima program, ki prepisuje `stdin` na `stdout`? Na videz nikakršnega. A spomnimo se, da zna `UNIX`-ova lupina preusmerjati datoteke, preden požene program. Recimo, ukaz

```
prog < data > res
```

požene program `prog` tako, da `stdin` zanj prihaja z datoteke `data`, `stdout` pa ne gre na ekran, pač pa na datoteko `res`. Če je `prog` naš program, smo datoteko `data` tako prepisali na datoteko `res`.

1.4.2 Vrednost prirejanja

Izurjeni `C` programerji bi napisali naš program drugače. Prirejanje, kot je

```
c = getchar()
```

ima tudi vrednost, namreč vrednost leve strani po prirejanju, in to vrednost lahko uporabimo v `while` testu:

```
#include <stdio.h>

/* copy stdin na stdout; second version */

int
main(void)
{
    int     c;

    while ((c = getchar()) != EOF)
        (void)putchar(c);

    return 0;
}
```

Ta verzija ima centraliziran vhod, samo enkrat smo napisali `getchar()`. Pri tem imamo na videz odvečne oklepaje, a niso odvečni. Če napišemo

```
c = getchar() != EOF
```

je to ekvivalentno konstrukciji

```
c = (getchar() != EOF)
```

to pa ni tisto, kar hočemo. Operator prirejanja `=` ima zelo nizko prioriteto, zato moramo prirejanje vedno vkleniti v oklepaje, če je prirejanje del večjega izraza.

1.4.3 Štetje znakov in vrstic

Kot naslednji primer programa bomo prešteli znake na datoteki `stdin`.

```
#include <stdio.h>

/* count characters in stdin */

int
main(void)
{
    long    nc;

    nc = 0;
    while (getchar() != EOF)
        ++nc;
    (void)printf("%ld\n", nc);

    return 0;
}
```

Vsakokrat, ko `getchar` vrne znak, ki ni `EOF`, je na `stdin` en znak več. Te znake štejemo v spremenljivki `nc`, ki je tipa `long`, ker utegne biti tip `int` preozek, recimo samo 16 bitov. Računanje s količino tipa `long` poteka kot s količino tipa `int`, le izpisovanje je drugačno. Formatno določilo `%ld` pravi, da je dodatni argument tipa `long` in ne `int`.

Podoben program je štetje vrstic. `ANSI C` poskrbi, da je vsaka (`ASCII`) datoteka videti kot zaporedje vrstic, od katerih se vsaka konča z znakom `'\n'`, torej ni treba nič drugega kot prešteti znake `'\n'`.

```
#include <stdio.h>

/* count lines in stdin */

int
main(void)
{
    int     c;
    long    nl;

    nl = 0;
    while ((c = getchar()) != EOF)
        if (c == '\n')
            ++nl;
    (void)printf("%ld\n", nl);

    return 0;
}
```

1.4.4 Štetje besed

Tretji program te vrste je štetje besed. Beseda je zaporedje znakov, ki niso beli znaki. Kaj so beli znaki, odloča standardna funkcija `isspace`[7], deklarirana v standardni glavi `<ctype.h>`. Približno rečeno pa so beli znaki `' '`, `'\n'`, `'\t'` in morda še kaj. Naš program bo oskubljena verzija `UNIX` programa `wc` (word count), ki izpiše število vrstic, število besed in število znakov v datoteki. To bo napravil tudi program `mywc`.

Prebrisani del je štetje besed. Med branjem smo "v besedi" ali pa "zunaj besede". Na začetku smo "zunaj besede". Vsak bel znak nas tudi prestavi v stanje "zunaj besede", vsak ne-bel znak pa nas prestavi v stanje "v besedi" in obenem pove, da smo našli novo besedo, če smo pravkar prestopili iz stanja "zunaj besede" v stanje "v besedi". Stanje bomo vodili v spremenljivki `state`. Njeni vrednosti bosta `IN` in `OUT`, dva predprocesorjeva makroja.

V programu je še zanimiva novost, večkratno prirejanje spremenljivkam `nc`, `nl` in `nw`. Več o tem bomo slišali kasneje. Sestavljen program A.8 je na strani 172.

1.4.5 Program count

Kot naslednji program vzemimo štetje posameznih kategorij znakov, recimo vsaka števka posebej, beli znaki vsi skupaj, in ostalo. Problem je privlečen za lase, ilustrira pa v enem kosu celo vrsto novih pojmov.

Ker imamo 12 kategorij znakov, je umestno vsaj za števke vpeljati večkratno vrednost. Tako lahko zgradimo program A.9 na strani 173. Rezultati, dobljeni s štetjem znakov v datoteki s programom samim, so

```
digits = 10 5 0 0 0 0 0 0 0 1, white space = 241, other = 433
```

Kaj še lahko povemo? Vsi števci, `ndigit[10]`, `nwhite`, `nother`, so tipa `long`, za vsak slučaj, če je tip `int` preozek.

Namenoma nismo uporabili funkcije `isdigit`, ki bi nam povedala, če je dani znak števka. To odločitev smo napravili sami s pogojem

```
if (c >= '0' && c <= '9')
```

To sicer funkcionira, a le pri pogoju, da uporabljamo kod, kjer so decimalne števke zakodirane kot zaporedna števila. V načelu je varneje uporabljati funkcijo `isdigit`. Res pa je, da tudi izraz

```
c - '0'
```

pomeni numerično vrednost števke, ki je shranjena v spremenljivki `c`, samo v primeru, da so števke zakodirane kot zaporedna števila.

Tudi funkcije `isspace` nismo uporabljali ampak smo rekli, da sami vemo, kateri znaki štejejo za bel znak: `' '`, `'\n'` in `'\t'` ter nihče več. Razlog, zakaj smo sami pisali te pogoje, je, da smo demonstrirali uporabo operatorjev `||` in `&&`. Operator `||` je logični "ali", operator `&&` pa logični "in". Njuna posebnost je v tem, da prevajalnik zgenerira kod, ki se preneha ukvarjati s tako sestavljenim pogojem tisti hip, ko je izid znan. V testu

---

[7] Pod `UNIX`-om glej `man isspace`

```
if (c == ' ' || c == '\n' || c == '\t')
```

je odločitev znana takoj, če je znak `c` res presledek in nima smisla testirati, če je mogoče enak znaku za novo vrsto ali predelčniku. V tem primeru je to samo optimizacija. V primeru

```
if (n < size && (buf[n] = getchar()) != EOF )
```

pa je več kot optimizacija. Če je res `n < size`, tedaj smemo prebrati naslednji znak, in ga shraniti v `buf[n]`, sicer pa ne, ker `buf[n]` ne obstaja več, če je `n >= size`.

Odločitev, v katero kategorijo spada znak `c`, sprejme kod

```
if (c >= '0' && c <= '9')
    ++ndigit[c - '0'];
else if (c == ' ' || c == '\n' || c == '\t')
    ++nwhite;
else
    ++nother;
```

To je poseben primer splošnejše konstrukcije

```
if (pogoj[1])
    stavek[1]
else if (pogoj[2)
    stavek[2]
...
else if (pogoj[n])
    stavek[n]
else
    stavek[n+1]
```

Če noben pogoj ni izpolnjen, se izvede `stavek[n+1]`, če obstaja. Če končni `else` in pripadajoči stavek opustimo, se v takem primeru ne zgodi nič.

V vsakem primeru je `stavek` lahko sestavljeni stavek v zavitih oklepajih.

Opisan stil pisanja je priporočljiv. Alternativa je, da vsakokrat zamaknemo `if` bolj v desno, kar ima za posledico, da pri komplicirani klobasi stavki odkorakajo čez desni rob papirja. Vrstice so pač omejene širine. Pogosto najdete že napisane programe, ki so jih napisali programerji, ki žive v veri, da je vrstica široka, kolikor se njim zdi potrebno. Do neke mere je to res, če imamo grafičen terminal, a tudi tam se enkrat konča. Iz pietete do uporabnikov, ki so vezani na alfanumerične terminale, pa ne predpostavljajte, da je lahko vrstica širša od 80 znakov.

Naloga 1.4. Preverite, če je vrednost izraza `getchar() != EOF` res nič ali ena.

Naloga 1.5. Sestavite program, ki bo natiskal vrednost `EOF`.

Naloga 1.6. Sestavite program, ki bo ločeno preštel presledke, predelčnike in nove vrste na standardnem vhodu.

Naloga 1.7. Sestavite program, ki bo prepisal standardni vhod na standardni izhod in pri tem nadomestil zaporedja več kot po enega presledka s po enim presledkom.

Naloga 1.8. Sestavite program, ki bo prepisal standardni vhod na standardni izhod in pri tem izpisal predelčnike kot `\t`, BACKSPACE znake kot `\b` in vsako nagibnico kot dve nagibnici. S tem postanejo, na primer, BACKSPACE zanki vidni.

Naloga 1.9. Kako bi preizkusili program `mywc`? Kakšni podatki bodo najverjetneje odkrili napake v programu, če so res v njem napake?

Naloga 1.10. Sestavite program, ki prepiše standardni vhod na standardni izhod, po eno besedo naenkrat, vsako v svojo vrsto.

Naloga 1.11. Sestavite program, ki bo natiskal histogram dolžin besed. To je lahka naloga, če so palčke histograma vodoravne, precej težja, če so palčke navpične.

Naloga 1.12. Sestavi histogram frekvenc različnih znakov na standardnem vhodu.

1.5 Funkcije

Funkcije so v `C` jeziku to, kar so funkcije in podprogrami v Fortranu ali Pascalu. Namen funkcije je, da vanjo skrijemo kakšno računanje in lahko potem tako funkcijo uporabimo, ne da bi skrbeli, kako je napisana.

Velikokrat bomo videli funkcijo, ki jo program pokliče samo enkrat. Njen namen je, da napravi program bolj pregleden. Funkcije v `C` jeziku so učinkovite in preproste, zato jih na veliko uporabljamo.

Funkcije smo uporabljali že doslej, a to so bile funkcije, ki jih je nekdo napisal za nas. Sedaj pa je že čas, da poskusimo napisati kakšno funkcijo tudi sami. Za vzorec vzemimo potenciranje. `C` nima operatorja potenciranja kot Fortran (`**`), ima pa funkcijo, `pow`[8], ki izračuna x^y za `double` argumenta `x` in `y`. Mi bomo sestavili močno poenostavljeno funkcijo, `power`, ki potencira celoštevilčno osnovo na nenegativen celoštevilčen eksponent, glej program A.10 na strani 174.

Definicija funkcije ima obliko

```
tip_vrnjene_vrednosti ime_funkcije(deklaracije_parametrov)
{
    definicije
    stavki
}
```

Funkcije so lahko definirane v poljubnem vrstem redu, v eni izvorni datoteki ali v večih. Če so definicije funkcij razmetane po več datotekah, je prevajanje in povezovanje bolj komplicirano, pisanje samo pa ne. Ene funkcije ne smemo deliti med več datotek.

Funkcija `main` kliče funkcijo `power` dvakrat, v vrstici

```
(void)printf("%d %d %d\n", i, power(2,i), power(-3,i));
```

Vsak klic poda funkciji `power` dva `int` argumenta, funkcija `power` pa vrne spet `int` vrednost, ki jo `printf` izpiše. Začetne vrstice definicije funkcije `power`

---

[8] Pod `UNIX`-om glej `man 3m pow`

```
static int
power(int base, int n)
```

deklarirajo tipe in imena parametrov ter tip rezultata. Parametra, `base` in `n`, sta za funkcijo `power` lokalni spremenljivki, ki jih klicatelj, `main`, inicializira na primerno vrednost. Parametra `base` in `n` ter definirani spremenljivki `i` in `p` so lokalni funkciji `power`. To pomeni, da zunaj funkcije `power` ta imena ne obstajajo, če pa že obstajajo, pomenijo nekaj popolnoma drugega.

Funkcija `power` je definirana kot statična funkcija. To pomeni, da ime funkcije `power` zunaj datoteke, kjer je definirana, ni poznano. To nam je všeč, če funkcije `power` zunaj datoteke `power.c` res nihče ne potrebuje.

Funkcija `power` vrne izračunano vrednost s stavkom `return`, ki ima obliko

```
return <izraz>
```

Funkcijam v splošnem ni treba vračati vrednosti, recimo funkcijam, ki so definirane, da vračajo `void`. Take funkcije uporabljajo `return` stavek brez `izraza`, ali pa se enostavno iztečejo v zaključni zaviti zaklepaj. Če pa je funkcija definirana, da vrača vrednost tipa `int`, mora to tudi storiti. Vrne lahko sicer tudi drugačen tip, če se ga da z implicitno prisilo predelati v tip `int`.

1.5.1 Funkcijski prototipi

Deklaracija na začetku datoteke

```
static int power(int base, int n);
```

se imenuje funkcijski prototip. Prototip navaja ime funkcije, njen tip in imena ter tipe njenih parametrov. Imena parametrov sicer niso potrebna, enakovreden prototip je

```
static int power(int, int);
```

a za dokumtacijo so imena parametrov pomembna in po možnosti naj bodo taka, da razumno opišejo, kaj parametri pomenijo.

Še malo zgodovine. **`K&R C`** še ni poznal funkcijskih prototipov. V **`K&R C`** bi definicija funkcije `power` zgledala takole

```
static int
power(base, n)
int base, n;
{
    int     i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p *= base;

    return p;
}
```

analogija prototipa pa

```
static int power();
```

`ANSI C` še vedno dovoljuje funkcije, napisane za `K&R C`, torej funkcije brez prototipov in brez tipov parametrov v definiciji. Specialen primer je funkcija brez parametrov, katere definicija je v `K&R C` videti takale

```
int func()
{
    ...
}
```

V `ANSI C` je zato treba funkcijo brez parametrov deklarirati tako, da se vidi, da nima parametrov, to je

```
int func(void);
```

ker deklaracija

```
int func();
```

implicira, da je to deklaracija v `K&R` stilu, ki lahko ima parametre ali pa ne.

Prototipi so zelo pomembni, ker omogočajo prevajalniku, da skrbneje preverja, če se klici in definicije funkcij ujemajo s prototipi.

1.5.2 Prenašanje parametrov

V `C` jeziku se vsi parametri funkcij prenašajo "po vrednosti". To pomeni, da funkcija dobi kopije dejanskih parametrov, ne pa dostopa do originalov, kot je to v Fortranu (vedno) ali v Pascalu (var parametri). Vsi parametri v jeziku `C` so vedno lokalne spremenljivke. Funkcija `swap`, ki naj bi zamenjala oba argumenta

```
void
swap(int a, int b)  /* NAPAK !!! */
{
    int     c;

    c = a;
    a = b;
    b = c;
}
```

to sicer stori, ampak na privatnih kopijah, ne pa na originalih, kot bi naiven programer, vajen Fortrana, pričakoval. Zato pa je po drugi strani parameter ravno prav inicializirana lokalna spremenljivka, torej lahko funkcijo `power` napišemo krajše:

```
static int
power(int base, int n)
{
    int     p;

    for (p = 1; n > 0; --n)
```

```
        p *= base;

    return p;
}
```

Spremenljivke `i` ne potrebujemo. Štejemo lahko kar v spremenljivki `n`, nazaj. Spremenljivka `n` ob začetku funkcije `power` že ima ravno pravo vrednost.

1.5.3 Najdaljša vrstica

Kot drugačen primer sestavimo program, ki bo prebral zaporedje vrstic in najdaljšo med njimi izpisal, torej

```
while (there is another line)
    if (it is longer than the previous longest)
        save it
        save its length
print the saved line
```

Ta načrt lahko precej naravno prepišemo v `C`. Sestavili bomo funkcijo `getline`, ki bo prebrala naslednjo vrstico. Ker želimo to funkcijo še kdaj uporabiti, jo bomo napisali kar se da splošno. Najmanj, kar zahtevamo od `getline` je, da nam pove, kdaj je prebrala zadnjo vrstico, natančneje, kdaj ni mogla prebrati še ene, ker je ni več. To dosežemo, če `getline` vrne dolžino vrstice, ki jo je prebrala. Ker ima vsaka vrstica vsaj en znak (znak za konec vrste), je dolžina nič razumen znak za konec podatkov.

Prebrano vrstico, če je dovolj dolga, moramo nekam shraniti, da jo bomo na koncu lahko izpisali. To nalogo bo opravila funkcija `copy`. Končno potrebujemo še program, ki bo povezoval vse skupaj. Izdelek je program A.11 na strani 175.

Za tekočo (in najdaljšo) vrstico moramo deklarirati večkraten znak, ki ima dovolj prostora za najdaljšo vrstico, ampak tega podatka pa nimamo. Zato se, na pamet, odločimo, da bo `1000` znakov dovolj in to vrednost priredimo predprocesorjevi spremenljivki, makroju, `MAXLINE`.

Preden začnemo pisati naše funkcije, deklariramo `getline` in `copy` s prototipnimi deklaracijami. Za tip vrnjene vrednosti smo rekli `static int` in `static void`. Statične funkcije eksistirajo samo v datoteki, kjer so deklarirane, njihova imena pa so zunaj datoteke popolnoma neznana. To so zaželene razmere, če vse funkcije uporabljamo samo v eni izvorni datoteki, kar je v tem primeru res. Funkcijo, ki jo uporabljamo v več izvornih datotekah, moramo deklarirati brez pridevnika `static`. Funkcije, ki niso statične, so globalne in so vidne v vseh izvornih datotekah.

Funkcijo getline smo definirali takole:

```
static int
getline(char s[], int lim)
```

Količina `s` je formalni parameter, torej zanjo ni treba napisati, koliko komponent ima. To moramo storiti samo tam, kjer tako vrednost definiramo, to je v funkciji `main` (v našem primeru). Povsod tam, kjer mora prevajalnik rezervirati kos pomnilnika, moramo napisati definicijo, ki v primeru večkratne vrednosti vključuje velikost. Če pa je pomnilnik že rezerviran, zadošča deklaracija in to je pri nas deklaracija parametra. Parameter `lim` pove, koliko prostora je v večkratni vrednosti `s`. To v resnici vemo, namreč `MAXLINE`, a v skladu s politiko, da hočemo napisati kar se da splošno funkcijo, nočemo vanjo "zavariti" imen kot je `MAXLINE`, ali pa celo konstant kot je `1000`.

Pogoj v stavku `for` je kompliciran:

```
i < lim - 1 && (c = getchar()) != EOF && c != '\n'
```

Količina `i` meri, koliko znakov je že prebranih. Če je `i` manjši od `lim - 1`, je prostora vsaj še za dva znaka, tistega, ki ga bomo prebrali, in `\0`. Samo tedaj preberemo znak `c` in če obstaja (ni `EOF`), pogledamo še, če ni znak za konec vrste, `'\n'`. Zanka se torej konča,

- če je `i` enak `lim - 1`; tedaj ne smemo prebrati še enega znaka , ker ga nimamo kam shraniti, saj moramo preostali nezaseden znak hraniti za prazen znak;
- če ni naslednjega znaka; tedaj ni kaj prebrati, zagotovo pa lahko shranimo še prazen znak;
- če je prebrani znak `'\n'`; v tem primeru znak `'\n'` shranimo in se potem delamo, da ni nobenega znaka več.

Vrstico zaključimo z `'\0'`, da bo funkcija `copy` znala ugotoviti, kako dolga je vrstica, ne da bi dodajali še en parameter, dolžino.

O funkciji `copy` ni kaj dosti povedati. V zanki prepišemo znak `from[i]` v `to[i]` in če prepisan znak ni `'\0'`, teče zanka naprej. Zanka se torej konča, ko prepiše znak `'\0'`. Jedro stavka `for` je prazen stavek. Funkcija `copy` se izteče v zaviti zaklepaj. Stavek `return` ni potreben, saj je funkcija deklarirana kot `void`, torej ne vrača vrednosti. Učinek bi bil isti, če bi pred zavitim zaklepajem napisali še `return` brez vrednosti.

1.6 Dolžina funkcij

Pri pisanju funkcij je treba paziti, da ne sestavljamo absurdno dolgih funkcij. Kot vodilo naj služi podatek, da je dvajset vrstic dolga funkcija že dolga.

Resnično vodilo je preglednost. Napisana funkcija mora biti pregledna, da jo lahko z očesom celo zajamemo. Od tu omejitev na dvajset vrstic, saj je na normalnem ASCII terminalu prostora za 24 vrstic.

P. J. Plauger v svoji knjigi *The Standard C Library* piše vse funkcije tako, da gre vsaka na dve strani, na sodo in liho stran, tako da ima bralec obe strani pred očmi. To je absoluten maksimum za velikost funkcije. Če je funkcija obsežnejša, moramo obračati strani sem ter tja kar pomeni, da napisana funkcija ni več pregledna.

Administrativna omejitev, naj bo vsaka funkcija napisana v največ dvajsetih vrsticah, je seveda nesmisel. Funkcija naj bo napisana pregledno, čeprav to znese trideset ali štirideset vrstic. S formalnimi zahtevami žal ne moremo prepričati slabega programerja, da bo pisal dober in razumljiv kod.

Naloga 1.13. Napišite program, ki bo izpisal vse vrstice na standardnem vhodu, ki so daljše od 80 znakov.

Naloga 1.14. Sestavite program, ki bo na koncu vrstic zbrisal presledke in eliminiral popolnoma prazne vrstice.

Naloga 1.15. Sestavite funkcijo `reverse`, ki obrne niz `s` na mestu. Uporabite to funkcijo, da bi obrnili standarden vhod po eno vrstico naenkrat.

Naloga 1.16. Sestavite program `detab`, ki bo vsak predelčnik na standardnem vhodu nadomestil s takim številom presledkov, da bo standardni izhod videti kot neprocesiran standardni vhod. Predpostavite, da so tabulatorji postavljeni na vsakih `n` znakov. Ali naj bo `n` spremenljivka ali makro?

Naloga 1.17. Sestavite program `entab`, ki bo zaporedne presledke zamenjal s predelčnikom tako, da bo izdelek videti kot prej. Program naj uporablja iste tabulatorje kot program `detab`.

Naloga 1.18. Programa `entab` in `detab` zlepite v en program, ki bo delal kot `detab` ali kot `entab`, v odvisnosti od imena, s katerim ga pokličemo. S tem je zagotovljeno, da bosta `entab` in `detab` res uporabljala iste tabulatorje.

Naloga 1.19. Sestavite `myfold`, ki prepogne predolge vrstice. Vrstica je je predolga, če je daljša od `n` znakov. Predolgo vrstico moramo prepogniti, če se da, med besedami, v nobenem primeru pa nove vrstice ne smejo biti daljše od `n` znakov. Od kod dobimo `n`? Kaj storiti, če srečamo besedo, daljšo od `n` znakov?

Naloga 1.20. Sestavite program, ki bo iz `C` programa odstranil vse komentarje. Ne pozabite na nize. Komentarji v jeziku `C` niso gnezdeni. Tak program je smislen, če imamo že napisan program z zanikrnimi komentarji in se odločimo, da delajo več škode kot koristi. To se zelo pogosto dogaja, če programerji pišejo komentarje zato, ker jim nekdo ukaže in ne zato, ker imajo kaj povedati. Ponekod plačujejo programerja od vrstice in v takih razmerah bomo dobili programe z zelo mnogo komentarjev, saj je komentarje laže pisati kot programe.

Naloga 1.21. Sestavite program, ki bo v `C` programih iskal osnovne sintaktične napake, kot so neuravnoteženi oklepaji, takšni ali drugačni. Ne pozabite na ubežna zaporedja in komentarje. Ta naloga je težka, če jo hočete rešiti v vsej splošnosti.

Poglavje 2

Tipi, operatorji in izrazi

V poglavju 1 smo si po hitrem postopku ogledali skoraj ves jezik `C`. V preostalih poglavjih si bomo vsebino poglavja 1 še enkrat ogledali, a to pot z vsemi podrobnostmi.

2.1 Konstante

Kako pišemo celoštevilske konstante? Konstanta je točno določenega tipa, ki je razviden iz oblike konstante. Tipičen primer je `1234`, ki je konstanta tipa `int`. Kako pa napišemo konstanto tipa `long`? Ali je vprašanje sploh pomembno? Je, ker v primeru mešanih tipov, `int` in `long`, `int` in `float`, `float` in `double`, `double` in `long double` in tako naprej, prevajalnik pretvori ožjo vrednost v siršo in to je tudi tip rezultata. Ponavadi nam je vseeno, če prevajalnik pretvarja tipe sem ter tja, včasih pa hočemo točno vedeti, kaj se dogaja, torej moramo za vsako konstanto znati povedati, kakšnega tipa je.

2.1.1 Cela števila

Običajna celoštevilčna konstanta, kot je `1234`, je tipa `int`. Če je tako velika, da je ne moremo shraniti v spremenljivko tipa `int`, tedaj je tipa `long`. Drug način, kako dosežemo, da je konstanta zagotovo tipa `long`, je, da dodamo na koncu `l` (ali `L`), torej `1234L` je tipa `long`, čeprav je `1234` tipa `int` (, če je `int` vsaj 16 bitov širok).

Če številu sledi `u` ali `U`, je število nepredznačenega tipa, `unsigned int`, ali na kratko, `unsigned`, recimo `1234U`. Če pa uporabimo pripono `UL` (ali `ul`), dobimo število tipa `unsigned long`, na primer, `1234UL`.

2.1.2 Desetiški, osmiški in šestnajstiški zapis

Ne glede na pripone `L`, `U`, `UL`, ali njihove male ekvivalente, je vsako celo število lahko zapisano v desetiškem sistemu (običajno), ali v osmiškem sistemu (osnova 8) ali pa v šestnajstiškem sistemu (osnova 16). Število je zapisano v osmiškem sistemu, če se začenja z ničlo, `01234`, in v šestnajstiškem sistemu, če se začenja z `0x` (ali `0X`), recimo `0x1234`. V šestnajstiškem sistemu igrajo črke `'A'`, `'B'`, `'C'`, `'D'`, `'E'` in `'F'` (ali njihovi mali ekvivalenti) vlogo "števk" `10`, `11`, `12`, `13`, `14` in `15`. Tako so `0377`, `0xff`, `0Xff`, `0xFF`, `0XFF` različni načini, kako zapišemo (desetiško) število `255`.

2.1.3 Znaki

Znaki (`char`) so v resnici majhna števila, odvisna od koda, ki ga uporabljamo. Znak `'A'` je drugačno ime za število `65` (, če uporabljamo `ASCII` znakovni kod). Količina `'A'` je torej tipa `char`, en zlog. Skonstruiramo pa lahko tudi poljubne osembitne vrednosti tipa `char` s konstrukcijo `'\ooo'` kjer so `ooo` ena, dve ali tri osmiške števke, torej števke med nič in sedem (vključno). Tako je na primer `'\101'` je (neobičajno) ime za znak `'A'` (v `ASCII` kodu). Isti učinek dosežemo z uporabo šestnajstiškega sistema, če pišemo `'\x41'` ali `'\X41'`, kar je spet neobičajno ime za znak `'A'` (v `ASCII` kodu). Znak `'\x0A'` je na primer (v `ASCII` kodu) znak za novo vrsto. Tako lahko pišemo

```
#define BELL    '\007'      /* ASCII bell character */
#define VTAB    '\013'      /* ASCII vertical TAB */
```

Večino takih znakov pa lahko pišemo še drugače, z ubežnimi zaporedji. Tu je popoln seznam ubežnih zaporedij v jeziku `C`:

`\a`	alert (bell)	`\\`	nagibnica
`\b`	backspace	`\?`	vprašaj
`\f`	nova stran	`\'`	enojni narekovaj
`\n`	nova vrsta	`\"`	dvojni narekovaj
`\r`	povratek voza	`\ooo`	osmiško število
`\t`	predelčnik	`\xhh`	šestnajstiško število
`\v`	vertikalni predelčnik		

Specialen primer ubežnega zporedja `'\ooo'` je ubežno zaporedje `'\0'`. To je seveda število nič, zapisano kot znak, `char`. Znak `'\0'` je prazen znak, ki služi kot oznaka konca niza (string).

Kot smo že omenili, `C` ne predpisuje, ali so količine tipa `char` predznačene ali ne. Tako na primer ne vemo, ali je `'a'` predznačeno število (znak) ali pa nepredznačeno. V tem primeru je seveda vseeno, ni pa vseeno v primeru `'\240'`, ali, kar je isto, `'\xa0'`.

2.1.4 Realna števila

Števila, ki vsebujejo decimalno piko in/ali decimalni eksponent, so tipa `double`: `3.14`, `2.71828e-4`, `1E-10`, `1e7`. Če takemu številu dodamo `f` (ali `F`), postane število tipa `float`: `3.14f`, `2.71828e-4f`, `1E-10F`, `1e7f`. Če pa dodamo `l` (ali `L`), postane število tipa `long double`: `3.14l`, `2.71828e-4l`, `1E-10L`, `1e7l`.

2.1.5 Konstantni izrazi

Izraz, v katerem so vsi sestavni deli konstante, se imenuje konstanten izraz. Misel je, da vrednost konstantnega izraza lahko izračuna že prevajalnik in to s pridom uporabljamo:

```
#define MAXLINE   1000
char line[MAXLINE + 1];
```

ali

```
#define LEAP    1       /* In a leap year */
int days[31+28+LEAP+31+30+31+30+31+31+30+31+30+31];
```

Definicija večkratne vrednosti namreč terja, da je število elementov v večkratni vrednosti konstantno, da že prevajalnik ve, koliko elementov ima večkratna vrednost.

2.1.6 Konstantni nizi

Konstanten niz je niz znakov v dvojnih narekovajih:

```
"Jaz sem niz"
```

ali

```
""      /* prazen niz */
```

V konstantnem nizu lahko uporabljamo ista ubežna zaporedja kot pri znakih. Prevajalnik bo med prevajanjem staknil dva konstantna niza, ki sledita drug drugemu, v en konstanten niz, zato lahko pišemo

```
"Beginning of a long string "
"that continues to the second line"
```

Ta prijem je potreben, ker mora biti niz zaključen v vrsti, kjer je začet. Uporabljamo lahko seveda tudi nagibnico na koncu vrste (, kot to lahko počnemo v Kornovi lupini[1]), vendar je stik nizov primernejši, ker lahko vmes potaknemo komentarje in presledke kot želimo.

Konstanten niz je tehnično večkraten znak, ki ima na koncu vedno prazen znak `'\0'`. Dolžina niza ni omejena, zato pa morajo programi sami ugotoviti, kje se niz konča. To delo opravlja standardna funkcija `strlen`[2]), deklarirana v standardni glavi `<string.h>`. Tu je naša verzija:

```
/* mystrlen: return length of s */
size_t
mystrlen(const char s[])
{
    size_t  i;

    i = 0;
    while (s[i] != '\0')
        ++i;

    return i;
}
```

Vrednost funkcije je tipa `size_t`, ki je celoštevilčen tip, dovolj širok, da lahko shrani vsako vrednost, ki jo lahko vrne operator `sizeof`. Tip `size_t` je deklariran v glavah `<stdio.h>`, `<stddef.h>`, `<stdlib.h>` in še v nekaterih standardnih glavah, tako da vsak program najbrž že vključuje dovolj standardnih glav, da je tip `size_t` že definiran.

---

[1] Pod **UNIX**-om glej `man ksh`

[2] Pod **UNIX**-om glej `man strlen`

2.1.7 Enumeracije

Obstaja še ena vrsta konstant, enumeracije, katerih obstoj je izposojen iz Pascala in zato nikoli niso dobile prave državljanske pravice v `C`. Vseeno, enumeracije obstajajo. Deklariramo jih na primer takole:

```
enum boolean { NO, YES };
```

Prvo navedeno ime, `NO` v našem primeru, dobi vrednost nič, naslednje ime, `YES` v našem primeru, dobi vrednost ena, in tako naprej.

```
enum escape { BELL = '\a', BACKSPACE = '\b', TAB = '\t',
              NEWLINE = '\n', VTAB = '\v', RETURN = '\r' };
```

V tem premeru smo za vsako ime navedli njegovo vrednost. Če navedemo samo nekatere vrednosti

```
enum months { JAN = 1, FEB, MAR, APR, MAY, JUN,
              JUL, AUG, SEP, OCT, NOV, DEC };
```

tedaj dobe nedefinrana imena vrednosti, ki slede, `FEB = 2`, `MAR = 3` in tako naprej.

Deklariramo lahko tudi spremenljivke enumeracijskih tipov, recimo,

```
enum boolean flag;
enum escape esc1, esc2;
enum months month = OCT;
```

a pravega pomena to nima, saj prevajalniki, takšni kot danes obstajajo, ne kontrolirajo, ali je vrednost, shranjena v spremenljivko, v skladu z enumeracijskim tipom: prevajalnik ne bo protestiral pri prirejanju

```
flag = 10;
```

Vseeno so enumeracije često boljše kot makroji (`#define`), saj prevajalnik lahko šteje namesto nas, ne glede na to, da bodo prevajalniki prihodnosti najbrž kontrolirali tudi, če je vrednost, shranjena v enumeracijsko spremenljivko, res takšnega tipa.

Naloga 2.1. Sestavite program, ki bo izpisal intervale vrednosti spremenljivk tipa `char`, `short`, `int` in `long`, vse predznačeno in nepredzančeno, s tem da prepišete vrednosti iz standardnih glav in z izračunom. Izračun je seveda mnogo težji. Ista naloga za realne tipe. To je sploh težka naloga.

2.2 Definicije

2.2.1 Definicije spremenljivk

Vsako spremenljivko moramo definirati preden jo uporabimo. Definicija pove, kakšnega tipa bodo spremenljivke, ki slede tipu. Recimo, definicije

```
int     money, n;
char    c, line[1000];
```

povedo, da sta `money` in `n` spremenljivki tipa `int`, spremenljivka `c` tipa `char`, spremenljivka `line` pa tisočkratni `char`.

Spremenljivke lahko razmečemo med definicije kot želimo. Tako lahko gornje definicije napišemo tudi kot

```
int     money;
int     n;
char    c;
char    line[1000];
```

Poslednji način ima to prednost, da lahko vsako definicijo posebej opremimo s komentarjem, ki pove, kaj spremenljivka pomeni. Seveda je pa še lepše, če že ime spremenljivke to pove.

2.2.2 Začetne vrednosti spremenljivk

Spremenljivkam lahko ob definiciji priredimo še začetne vrednosti, recimo

```
char    esc = '\\';
int     i = 0;
int     limit = MAXLINE + 1;
float   eps = 1.0e-5;
```

2.2.3 Statične in avtomatične spremenljivke

Spremenljivke, ki so definirane v notranjosti funkcije, so praviloma avtomatične, spremenljivke, ki so definirane zunaj vseh funkcij pa so vedno statične. Avtomatičnost pomeni, da je spremenljivka shranjena na skladu in da spremenljivka nastane ob vhodu v funkcijo, ko pa se funkcija vrne, njene avtomatične spremenljivke izginejo. Pogosto govorimo, da funkcija ob klicu dodeli prostor za svoje spremenljivke na skladu. Tako izražanje, čeprav je formalno pravilno, je nevarno, ker si programerji pogosto napak predstavljajo, kaj pomeni dodeljevanje. V tem primeru beseda dodeljevanje nima nobene zveze s funkcijo `malloc`, ki jo bomo še spoznali in ki tudi dodeli kos pomnilnika, samo ne na skladu.

Začetne vrednosti statičnih spremenljivk morajo biti konstantne. To pomeni, da mora že prevajalnik (skorajda) vedeti, kakšne so te začetne vrednosti, ker se te začetne vrednosti shranijo v statične spremenljivke samo enkrat, v principu še preden se začne program izvajati. Če statična spremenljivka nima definirane začetne vrednosti, je ta začetna vrednost nič, natančneje, vsi biti take vrednosti so nič.

Avtomatične spremenljivke živeo na skladu in se rode ob klicu funkcije. To pomeni, da ob klicu funkcije avtomatična spremenljivka nima definirane vrednosti. Po drugi strani pa prevajalnik dopušča, da sami predpišemo začetne vrednosti avtomatičnih spremenljivk, ki so lahko poljubnji izrazi. To seveda pomeni, da v funkciji med konstrukcijo

```
int     num = 10;
```

in konstrukcijo

```
int     num;

num = 10;
```

ni nobene razlike, ker mora v obeh primerih prevajalnik skonstruirati ukaze, ki na izbrano mesto na skladu (spremenljivka `num`) shranijo vrednost `10`.

2.2.4 Kvalifikacija const

Definicija spremenljivke je lahko opremljena še s kvalifikacijo `const`, recimo

```
const double e = 2.7182845905;
```

ki pove, da se vrednost take spremenljivke ne bo spreminjala. Kaj bo prevajalnik v takem primeru napravil, je prepuščeno sestavljalcem prevajalnikov, vsaj zaenkrat. Vsekakor pa je pametno povedati, da je spremenljivka konstantna, če vemo, da je res tako. Prej ali slej bo kakšen prevajalnik protestiral, če bomo hoteli konstantno spremenljivko spreminjati, ne glede na to, da utegne prevajalnik tako spremenljivko shraniti v kos pomnilnika, v katerega se ne da pisati in bo računalnik to ugotovil med izvajanjem.

Če za večkratno vrednost rečemo, da je konstantna, recimo

```
const char msg[] = "warning: ";
```

to pomeni, da so posamezne komponente konstantne, da komponent ne bomo spreminjali.

2.3 Operatorji

2.3.1 Aritmetični operatoji

`C` ima zelo veliko operatorjev. Binarni aritmetični operatorji, to je, operatorji z levim in desnim argumentom, so

`+`	seštevanje
`-`	odštevanje
`*`	množenje
`/`	deljenje
`%`	ostanek pri deljenju

Deljenje dveh števil celoštevilčnega tipa nam dá celoštevilčen kvocient, zaokrožen proti ničli. Pri takem deljenju torej izgubimo ostanek, ki ga pa lahko eksplicitno izračunamo z operatorjem `%`. Izraz

```
x % y
```

nam dá ostanek pri deljenju `x` z `y`, torej nič, če se deljenje izide.

Primer. Po (Gregorijanskem) koledarju je vsako četrto leto prestopno z izjemo vsakega stotega leta, ki ni prestopno, z izjemo vsakega štiristotega leta, ki vendarle je prestopno. Torej

```
if (year % 4 == 0 && year % 100 != 0 || year % 400 == 0)
    (void)printf("%d is a leap year\n", year);
else
    (void)printf("%d is not a leap year\n", year);
```

2.3.2 Prioritete in asociativnost

Binarna operatorja `+` in `-` imata isto prioriteto, ki je nižja od prioritete binarnih operatorjev `*`, `/` in `%`, ki je spet nižja od prioritete vseh unarnih operatorjev, to je, operatorjev samo z enim argumentom. Prioritete operatorjev so nastavljene tako, da je zgornji primer pravilen, čeprav ni najbolj jasen. Operatorja `%` ne moremo uporabiti na realnih (`float`, `double`, `long double`) argumentih. Kaj se zgodi pri prekoračitvi (ali podkoračitvi) obsega, je prepuščeno računalniku.

Če prioriteta ne razsodi, kateri operator je prej na vrsti, tedaj razsodi asociativnost: vsi aritmetični operatorji asociirajo z leve proti desni.

2.3.3 Relacijski operatorji

Relacijski operatorji

<	manjši
>	večji
<=	manjši ali enak
>=	večji ali enak

imajo vsi isto prioriteto in asociirajo z leve proti desni, kar pa ni zelo koristno, saj `x < y < z` ne pomeni tega, kar to pomeni v matematiki: `x < y && y < z`, ampak `(x < y) < z`

Po prioriteti tik pod njimi sta relacijska operatorja

==	enak
!=	neenak

ki spet asociirata z leve proti desni (ni koristi). Relacijski operatorji imajo nižjo prioriteto kot aritmetični operatorji, zato izraz

```
i < lim - 1
```

pomeni to, kar si mislimo, namreč

```
i < (lim - 1)
```

Operator `&&` ima višjo prioriteto kot operator `||`, ti prioriteti pa sta nižji kot prioritete relacijskih operatorjev, zato izraz

```
i < lim - 1 && (c = getchar()) != '\n' && c != EOF
```

ne potrebuje dodatnih oklepajev.

Vrednost relacije je po definiciji ena, če je pogoj izpolnjen, in nič, če ni. Unaren operator `!` pretvori neničelen argument v nič in ničelnega v ena. Tipično uporabljamo operator `!` v situacijah kot so

```
if (!valid)
```

namesto

```
if (valid == 0)
```

Odločitev, katera oblika je bolj čitljiva, pa prepuščamo bralcu.

2.3.4 Povečevanje in zmanjševanje

`C` ima dva neobičajna operatorja za povečevanje in zmanjševanje spremenljivk, torej za prištevanje (odštevanje) enke, na primer

```
if (c == '\n')
    ++nl;
```

Operatorja `++` in `--` lahko uporabimo kot predponska operatorja (`++nl`) ali kot priponska operatorja (`nl++`). V obeh primerih se vrednost spremenljivke `nl` poveča za ena, razlika pa je v vrednosti izraza samega: `++nl` je vrednost spremenljivke `nl` po povečevanju, `nl++` pa vrednost spremenljivke `nl` pred povečevanjem. Enako razmišljanje velja za operator `--`.

V kontekstih, kjer nas vrednost spremenljivke ne zanima, kjer potrebujemo samo učinek povečevanja ali zmanjševanja, na primer

```
if (c == '\n')
    ++nl;
```

je vseeno, ali uporabimo predponsko obliko ali priponsko obliko. Včasih pa moramo uporabiti natanko pravo obliko. Oglejmo si na primer funkcijo

```
void squeeze(char s[], const int c);
```

ki iz niza `s` izcedi vse primerke znaka `c`. Tu je izdelek

```
/* squeeze: delete all c from s */
void
squeeze(char s[], const int c)
{
    int     i, j;

    for (i = j = 0; s[i] != '\0'; ++i)
        if (s[i] != c)
            s[j++] = s[i];
    s[j] = '\0';
}
```

Vsakokrat, ko naletimo na znak, ki ni enak `c`, tak znak prepišemo v `s[j]` in šele potem povečamo `j`, da bo pripravljen za naslednji znak.

Podobna naloga je funkcija

```
void mystrcat(char s[], const char t[]);
```

ki prepiše niz `t` na konec niza `s`. Funkcijo imenujemo `mystrcat`, ker v standardni knjižnici že obstaja funkcija `strcat`[3]) z isto nalogo, le da sta parametra kazalca, funkcija sama pa vrne kazalec na začetek zgrajenega niza.

```
/* mystrcat:
 * concatenate t to end of s; s must be big enough
 */
```

---

[3] Pod `UNIX`-om glej `man strcat`

```
void
mystrcat(char s[], const char t[])
{
    int     i, j;

    i = j = 0;
    while (s[i] != '\0')                /* find end of s */
        ++i;
    while ((s[i++] = t[j++]) != '\0')   /* copy t */
        ;
}
```

2.3.5 Bitni operatorji

`C` pozna tudi operatorje za manipulacije z biti. Te operatorje lahko uporabimo le na celoštevilčnih operandih, to je `char`, `short`, `int` in `long`, bodisi predznačenega bodisi nepredznačenega okusa. Ti operatorji so

`&`	IN po bitih
`|`	inkluziven ALI po bitih
`^`	ekskluziven ALI po bitih
`<<`	pomik v levo
`>>`	pomik v desno
`~`	eniški komplement po bitih (unaren operator)

Operator `&` pogosto uporabljamo, da iz števila izluščimo samo izbrane bite, recimo

```
n = n & 0177;
```

obdrži v spremenljivki `n` samo desnih sedem bitov, ostali postanejo enaki nič. Primer za tako operacijo je pretvorba poljubnega števila v sedembitni `ASCII` znak, pretvorba v smislu, da je rezultat garantiran sedembitni `ASCII` znak.

Operator `|` uporabimo za ”prižiganje bitov”:

```
x = x | SET_ON;
```

”prižge” vse tiste bite v spremenljivki `x`, ki so ”prižgani” v `SET_ON`.

Operator `^` ”prižge” v rezultatu tiste bite, ki so v obeh argumentih različni.

Pri bitnih operatorjih `&` in `|` je treba biti previden, da jih ne zamenjamo z logičnima operatorjema `&&` in `||`, ki pomenita izračun logične vrednosti od leve proti desni. Če je `x` enak `1`, `y` pa `2`, tedaj je `x & y` enak nič, `x && y` pa ena.

Operatorja `<<` in `>>` pomenita levi oziroma desni pomik levega operanda za število bitov predpisanih z desnim operandom, ki mora biti nenegativno število. Levi pomik za `-2` ni desni pomik za dva ampak napaka. Levi pomik vstavlja na izpraznjena mesta ničle, torej je levi pomik za dva enakovreden množenju s štiri (, če število ni preveliko). Desni pomik nepredznačene količine spet vstavi na izpraznjena mesta (na levi strani) ničle. Desni pomik predznačene količine pa na izpraznjena mesta lahko vstavi ničle (logični pomik) ali kopije prvega bita (predznak, aritmetični pomik), odvisno od arhitekture računalnika. Nauk: desni pomik uporabljamo samo na nepredznačenih operandih.

Unarni operator `~` nam da eniški komplement števila, to je, pretvori vsak bit nič v bit ena, vsak bit ena pa bit nič. Na primer,

```
x = x & ~077
```

"ugasne" desnih šest bitov. Izraz `x & ~077` je neodvisen od širine celih števil in nam je ljubši od `x & 0177700`, ki je pravilen le na računalnikih, ki imajo šestnajstbitna cela števila.

Kot primer si oglejmo funkcijo

```
unsigned getbits(const unsigned x, const int p, const int n);
```

ki iz količine `x` izreže `n` bitno polje, ki se začenja na poziciji `p`. Dobljeno polje naj bo desno poravnano. Pri tem je pozicija nič desni rob. Parametra `p` in `n` naj imata smiselne vrednosti. Tako naj na primer `getbits(x, 5, 3)` vrne tri bite s pozicij `5`, `4` in `3`, desno poravnane v rezultat, recimo, `getbits(0345, 5, 3)` vrne `4`.

```
/* getbits: get n bits from position p */
unsigned
getbits(const unsigned x, const int p, const int n)
{
    return (x >> (p + 1 - n)) & ~(~0U << n);
}
```

Izraz `p + 1 - n` pove, kako daleč na desno moramo pomakniti vse bite. Zviti del je maska, ki iz pomaknjenega števila izreže desnih `n` bitov. Količina `~0U` je maska iz samih enojk, toliko enojk kot, ima `unsigned` bitov; `~0U << n` je maska iz enojk, ki ima na koncu `n` ničel, njen komplement pa je maska, ki jo potrebujemo.

2.3.6 Operatorji s prirejanjem

Zanimivi so operatorji s prirejanjem. Izraz kot je

```
i = i + 2
```

kjer se spremenljivka na levi ponovi na desni, lahko poenostavimo v

```
i += 2
```

Za večino binarnih operatorjev (to je operatorjev, ki imajo levi in desni argument), obstaja ustrezen operator s prirejanjem. Če je `op` tak operator, je `op=` operator s prirejanjem. Operatorji, iz katerih lahko z enačajem naredimo operatorje s prirejanjem, so

```
+   -   *   /   %   <<  >>  &   ^   |
```

Če sta `izraz1` in `izraz2` dva izraza, tedaj

```
izraz1 op= izraz2
```

pomeni

```
izraz1 = (izraz1) op (izraz2)
```

s pomembnim dodatkom, da se `izraz1` izračuna samo enkrat. Oklepaji okrog `izraz1` in `izraz2` so pomembni, ker

```
x *= y + 1
```

pomeni

```
x = x * (y + 1)
```

in ne

```
x = x * y + 1
```

Za vzorec napišimo funkcijo

```
int bitcount(unsigned x)
```

ki prešteje število enojk v binarni reprezentaciji števila `x`.

```
/* bitcount: count 1 bits in x */
int
bitcount(unsigned x)
{
    int     b;

    for (b = 0; x != 0; x >>= 1)
        if (x & 01)
            ++b;
    return b;
}
```

Argument je deklariran kot nepredznačeno število, da nismo odvisni od arhitekture računalnika.

Operatorji s prirejanjem imajo vrsto ugodnih lastnosti. V prvi vrsti ponazarjajo način, kako ljudje premišljujemo. Pravimo: "Prištej dva k stevilu `i`" in ne "Vzemi število `i`, prištej dva in shrani vsoto nazaj v `i`". Drug razlog je, da je komplicirane izraze kot je

```
yyval[yypv[p3+p4] + yypv[p1+p2]] += 2
```

lažje razumeti: ni treba mukoma primerjati dveh dolgih izrazov in ugibati, zakaj nista enaka, če sta pomotoma različna.

2.3.7 Pogojni izrazi

Zaporedje stavkov

```
if (a > b)
    z = a;
else
    z = b;
```

shrani v `z` večje od števil `a` in `b`. Isti učinek lahko dosežemo s stavkom

```
z = (a > b) ? a : b;
```

Konstrukciji

```
izraz1 ? izraz2 : izraz3
```

pravimo pogojni izraz. Računalnik najprej izračuna vrednost izraza `izraz1`. Če je neničelna (true), tedaj je vrednost pogojnega izraza vrednost izraza `izraz2`, v nasprotjen primeru pa vrednost izraza `izraz3`. Oklepaji okrog izraza `izraz1` v resnici niso potrebni, saj ima operator `?` zelo nizko prioriteto, običajno pa take oklepaje pišemo, ker je tak kod bolj pregleden.

Samo eden od izrazov `izraz2` in `izraz3` se res izračuna. Ob tem je treba poudariti, da oba izraza `izraz2` in `izraz3` sodelujeta pri določanju tipa pogojnega izraza. Če je izraz `f` tipa `float`, izraz `n` pa tipa `int`, je izraz

```
(n > 0) ? f : n
```

tipa `float` ne glede na to ali je `n` pozitivno število ali ne.

2.3.8 Pretvarjanje tipov

Pedantno vzeto, v jeziku `C` imajo binarni operatorji praviloma oba operanda istega tipa. Kadar ima (binaren) operator operanda različnih tipov, ju je treba prevesti na "skupni imenovalec", to je, poskrbeti, da sta oba operanda istega tipa in potem opraviti operacijo. To pretvarjanje se zgodi avtomatično in neopazno, praviloma s tem, da prevajalnik ožji operand razširi v širšega, na primer `int` v `float`. Operacije, ki nimajo smisla, na primer `float` indeks, so prepovedane. Transformacije širših tipov v ožje utegnejo sprožiti svarilo, napačne pa niso.

Ker so znaki, tip `char`, kar majhna (ozka) števila, je transformiranje znakov zelo poenostavljeno, še preveč, saj ponuja naivne izvedbe funkcij kot so `atoi`, ki pretvori niz števk v število. Glava `<stdlib.h>` vsebuje prototip prave funkcije `atoi`[4], naivna verzija `myatoi` pa je takale

```
/* myatoi: naive conversion of s to integer */
int
myatoi(const char s[])
{
    int     i, n;

    n = 0;
    for (i = 0; s[i] >= '0' && s[i] <= '9'; ++i)
        n = 10 * n + (s[i] - '0');

    return n;
}
```

---

[4] Pod `UNIX`-om glej `man atoi`

Funkcija `myatoi` je naivna, med drugim, zato, ker se zanaša, da decimalne števke tvorijo strnjeno zaporedje znakov v kodu, ki ga uporabljamo, kar ni treba, da je res.

Drug tak primer je funkcija `lower`, ki pretvori veliko črko v malo. Če argument ni velika črka, ostane nespremenjen. Glava `<ctype.h>` vsebuje prototip prave funkcije `tolower`[5], naivna verzija `lower` pa je

```
/* lower: naive conversion to lower case */
int
lower(int c)
{
    return (c >= 'A' && c <= 'Z') ? c + 'a' - 'A' : c;
}
```

Glava `<ctype.h>` vsebuje poleg funkcij `tolower` in `toupper` še vrsto predikatnih funkcij

`int isalpha(int c)`	znak `c` je črka
`int isupper(int c)`	znak `c` je velika črka
`int islower(int c)`	znak `c` je mala črka
`int isdigit(int c)`	znak `c` je desetiška števka
`int isxdigit(int c)`	znak `c` je šestnajstiška števka
`int isalnum(int c)`	znak `c` je črka ali desetiška števka
`int isspace(int c)`	znak `c` je bel znak: presledek, predelčnik, `NL`, `CR`, `VT`, `FF` in morda še kaj
`int ispunct(int c)`	znak `c` je ločilo
`int isprint(int c)`	znak `c` je izpisljiv (presledek je izpisljiv)
`int isgraph(int c)`	znak `c` ima grafično podobo (presledek je nima)
`int iscntrl(int c)`	znak `c` je kontrolni znak
`int isascii(int c)`	znak `c` je `ASCII` znak (kod med `0` in `0177` vključno)

Na primer, test

```
c >= '0' && c <= '9'
```

moremo nadomestiti s funkcijo `isdigit`[6].

Implicitna pretvarjanja delujejo kot pričakujemo: ”nižji” tip se pretvori v ”višjega”, operacija se opravi na ”višjem” tipu in tega tipa je tudi rezultat. Če ni nepredznačenih operandov, so pravila preprosta:

- Če je en operand `long double`, pretvori drugega v `long double`.
- Sicer, če je en operand `double`, pretvori drugega v `double`.
- Sicer, če je en operand `float`, pretvori drugega v `float`.
- Sicer, pretvori `char` in `short` v `int`.
- Koncno, če je en operand `long`, pretvori drugega v `long`.

---

[5] Pod `UNIX`-om glej `man tolower`

[6] Pod `UNIX`-om glej `man isdigit`

`ANSI C` ne razširi argumentov tipa `float` avtomatično v `double` kot `K&R C`. Matematične funkcije, deklarirane v glavi `<math.h>`, uporabljajo in vračajo tip `double`, razen funkcij, ki imajo imena, ki se končujejo na `f`, in ki delujejo s tipom `float`, na primer `sinf` je `float` verzija `double` funkcije `sin`.

Pretvarjanja so bolj zamotana, če so vpleteni nepredznačeni argumenti. Naslednji primer pokaze, kakšne sitnosti srečujemo. Če je `int` širok `16` bitov, `long` pa `32`, tedaj je `-1L < 1U`, ker je `1U` `unsigned int`, ki je promoviran v `signed long`, v tip konstante `-1L`; po drugi strani je `-1L > 1UL`, ker je `-1L` promoviran v `unsigned long`, v tip konstante `1UL`, ki je potem videti kot ogromno pozitivno število.

Do pretvarjanja tipov pride tudi pri prirejanju. Vrednost izraza na desni je treba prilagoditi tipu spremenljivke na levi.

Tip `char` se razširi v tip `int` tako, da so dodatni biti na levi ničle, v nekaterih izvedbah pa se prvi (levi) bit `char` vrednosti razširi na dodatne bite. Če imamo pretvarjanje tipa `char` v tip `int`, v resnici ne vemo, kaj se bo zgodilo. Edini zanesljiv slučaj tega pretvarjanja je pretvarjanje `unsigned char` v `int`, kjer se biti na levi dopolnijo z ničlami. Za običajen `char` pa žal ne vemo, ali je predznačen ali ne.

Širša cela števila se pretvorijo v ožja s tem, da odvržemo odvečne bite na levi, neodvisno od predznačenosti. Seveda utegne biti tako pretvarjanje nereverzibilno, če smo res odvrgli kaj dobrih bitov. Primer:

```
int     i;
char    c;

i = c;
c = i;
```

Vrednost znaka `c` ostane nespremenjena. Če pa zamenjamo vrsti red prirejanj, vrednost `i` morda ni več tista, s katero smo začeli.

Če je `x` tipa `float`, `i` pa tipa `int`, tedaj obe prirejanji, `x = i` in `i = x`, pomenita pretvarjanje. Pretvarjanje `float` v `int` pomeni izgubo necelega dela. Tudi pretvarjanje `int` v `float` ni nujno reverzibilno. Količina tipa `int` ima lahko toliko bitov, da ne morejo biti vsi shranjeni v količini tipa `float`. Pri pretvarjanju `double` v `float` je odvisno od izvedbe, ali bo rezultat zaokrožen ali ne.

Argumenti funkcij so izrazi in zato prihaja do pretvarjanja tudi pri podajanju argumentov funkciji (in pri podajanju rezultata klicatelju). Če prototipna deklaracija ni vidna, v `K&R C` jih sploh ni, torej niso vidne, tedaj bo prevajalnik pretvoril `char` in `short` v `int` ter `float` v `double`.

Končno lahko izsilimo pretvarjanje tudi z unarnim operatorjem prisile: v konstrukciji

```
(ime-tipa)izraz
```

se vrednost izraza `izraz` pretvori v tip, identificiran z `ime-tipa`, na primer

```
(float)n
(int)x
```

Funkcije, deklarirane v glavi `<math.h>`, imajo večinoma argumente tipa `double` in vračajo vrednosti tipa `double`. Če je `n` tipa `int`, moramo torej pisati

```
sqrt((double)n)
```

ki pretvori `int n` v `double`, preden ga posreduje funkciji `sqrt`. Povedano velja za `K&R C`. V `ANSI C` uporabljamo prototipe, ki definirajo tipe argumentov, tako da prisile pri klicih niso potrebne, recimo

```
root2 = sqrt(2);    /* ANSI C,
                     * deklaracija prototipa
                     * sqrt vidna
                     */
```

2.3.9 Slučajna števila

Standardna knjižnica vsebuje še generator slučajnih stevil, `rand`[7], skupaj s funkcijo za definicijo semena, `srand`[8]. Generator sam je precej zanikrn[9], vendar je tako preprost, da ga lahko napišemo

```
int rand(void);
void srand(unsigned seed);
static unsigned long next = 1;

/* rand: return pseudo-random integer on 0 .. 32767 */
int
rand(void)
{
    next = next * 1103515245 + 12345;
    return (unsigned)(next / 65536) % 32768;
}

/* srand: set seed for rand() */
void
srand(unsigned seed)
{
    next = seed;
}
```

Ker hočemo vedeti, kaj se dogaja, mora operator `%` delati z nepredznačenimi količinami.

2.3.10 Prioritete operatorjev in njihova asociativnost

Za konec sestavimo še tabelo vseh operatorjev skupaj z njihovo prioriteto in asociativnostjo. Operatorji v začetnih vrsticah imajo višjo prioriteto kot operatorji v kasnejših vrsticah. Operatorji v isti vrstici imajo isto prioriteto. Če v izrazu nastopata dva operatorja z iste vrstice, tedaj asociativnost pove, kako se vežeta.

---

[7] Pod `UNIX`-om glej `man rand`

[8] Pod `UNIX`-om glej `man srand`

[9] glej W. H. Press et al., Numerical Recipes in C, Cambridge University Press, 1994

OPERATORJI	ASOCIATIVNOST
`() [] -> .`	z leve na desno
`! ~ ++ -- + - * & (ime-tipa) sizeof`	z desne na levo
`* / %`	z leve na desno
`+ -`	z leve na desno
`<< >>`	z leve na desno
`< <= > >=`	z leve na desno
`== !=`	z leve na desno
`&`	z leve na desno
`^`	z leve na desno
`|`	z leve na desno
`&&`	z leve na desno
`||`	z leve na desno
`? :`	z desne na levo
`= += -= *= /= %= &= ^= |= <<= >>=`	z desne na levo
`,`	z leve na desno

Ta tabela povsem določa, kje v izrazu so implicitni oklepaji.

`C`, za razliko od drugih jezikov, ne predpisuje, v kakšnem vrstem redu je treba izračunati operande operatorjev. Izjeme so operatorji `&&`, `||`, `?`, `:` in `,`. V stavku kot je

```
x = f() + g();
```

ne moremo povedati, ali bo računalnik prej izračunal vrednost funklcije `f` ali vrednost funkcije `g`. Izbran vrsti red lahko izsilimo, če vrednost prve funkcije shranimo v kakšno vmesno spremenljivko.

Tudi funkcijski argumenti se izračunajo v nam neznanem vrstnem redu. Stavek

```
(void)printf("%d %d\n", ++n, power(2, n));  /* NAPAK */
```

je napačen. Nekateri prevajalniki (računalniki) bodo računali potenco z originalnim eksponentom, nekateri pa s povečanim, ker ne vemo, ali bo prevajalnik prej izračunal `++n` in šele potem `power(2, n)`, ali pa obratno.

Drug izvor nerodnih napak so stranski učinki, kjer se neka spremenljivka spremeni kot stranski učinek izračunavanja kakšnega izraza. Zelo neroden primer je stavek

```
a[i] = i++;
```

Dilema je, ali je indeks stara vrednost `i` ali povečevana. Standard takih reči namenoma ne predpisuje, to pa pomeni, da jih ne smemo uporabljati.

Naloga 2.2. Brez `&&` ali `||` operatorja sestavite zanko, ekvivalentno zanki

```
for (i=0; i < lim-1 && (c=getchar()) != '\n' && c != EOF; ++i)
    s[i] = c;
```

Naloga 2.3. Sestavite funkcijo `htoi`, ki pretvori zaporedje šestnajstiških števk (vključno z `0x` ali `0X`) v celo število. Dovoljene števke so `0` do `9`, `a` do `f` ali `A` do `F`.

Naloga 2.4. Sestavite novo verzijo funkcije

```
void squeeze(char s[], const char t[])
```

ki iz `s` odstrani vse znake, ki se ujemajo s katerimkoli znakom v `t`.

Naloga 2.5. Sestavite funkcijo

```
int any(const char s[], const char t[])
```

ki vrne indeks prvega znaka v `s`, ki je hkrati tudi v `t`. (Standardna glava `<string.h>` vsebuje deklaracijo funkcije `strpbrk`, ki opravlja isto nalogo, le da vrne kazalec namesto indeksa).

Naloga 2.6. Sestavite funkcijo

```
int setbits(const int x, const int p,
    const int n, const int y)
```

ki vrne število `x`, v katerem je `n` bitov, ki se začenjajo na poziciji `p`, nadomeščenih s spodnjimi `n` biti števila `y`.

Naloga 2.7. Sestavite funkcijo

```
int invert(const int x, const int p, const int n)
```

ki invertira (spremeni `1` v `0`, `0` v `1`) `n` bitov, ki se začenjajo na poziciji `p`. Ostali biti ostanejo nespremenljeni.

Naloga 2.8. Sestavite funkcijo

```
int rightrot(const int x, const int n)
```

ki vrne vrednost `x`, rotirano v desno za `n` bitov.

Naloga 2.9. Pokažite, da v dvojiškem komplementu izraz `x &= x - 1` zbriše najbolj desen bit `1` v stevilu `x`. Če je to res, napišite hitrejšo verzijo funkcije `bitcount`.

Poglavje 3

Kontrolni stavki

3.1 Stavek in blok

Izrazi kot so `x = 0`, `++i` ali `(void)printf(...)` postanejo stavki, če jim sledi podpičje:

```
x = 0;
++i;
(void)printf(...);
```

Podpičje v `C` zaključuje stavke, v Pascalu pa jih ločuje.

Z zavitimi oklepaji tvorimo sestavljene stavke, recimo:

```
{
    x = 0;
    ++i;
    (void)printf(...);
}
```

Če sestavljeni stavek vsebuje (na začetku) definicije, mu pravimo blok. Zaključnemu zavitemu oklepaju, ki končuje sestavljeni stavek (ali blok), nikoli ne sledi podpičje.

3.2 Pogojni stavek

S konstrukcijo

```
if (pogoj)
    stavek1
else
    stavek2
```

delamo odločitve. Če je vrednost izraza `pogoj` neničelna (true), tedaj se izvede `stavek1` sicer pa `stavek2`. Poenostavljena verzija

```
if (pogoj)
    stavek
```

pravi: če je vrednost izraza `pogoj` neničelna (true), tedaj se izvede `stavek` sicer pa nič.

Namesto pogoja

```
if (izraz != 0)
```

pogosto napišemo kar

```
if (izraz)
```

Oba načina sta enakovredna, a kateri je bolj berljiv, je prepuščeno okusu programerja.

Ker `else` del lahko manjka, naletimo na standardni problem, problem bingljajočega `else`. V konstrukciji

```
if (n > 0)
    if (a > b)
        z = a;
    else
        z = b;
```

je nejasno, kam spada `else`, k prvemu ali k drugemu `if`. Z zamikanjem smo sicer nakazali, kam mislimo, da `else` spada, vendar zamikanje prevajalniku nič ne pomeni. Pravilo je, da vsak `else` spada k najbližjemu `if`, torej zgornji primer res dela tako, kot nakazuje zamikanje.

Problem je lahko bolj skrit, recimo

```
if (n > 0)
    for (i = 0; i < n; ++i)
        if (s[i] > 0) {
            (void)printf("...");
            return i;
        }
else            /* NAPAK */
    (void)printf("error -- n is negative\n");
```

Zamikanje kaže, kaj smo hoteli doseči, prevajalnik pa ne sledi zamikanju in prevede program tako, kot je napisan in ne tako kot je zamaknjen.

Mimogrede, v primeru

```
if (a > b)
    z = a;
else
    z = b;
```

je na koncu stavka `z = a;` podpičje, čeprav mu sledi `else`. Ta `else` bi v Pascalu pomenil, da podpičje ni potrebno, v `C` pa mora biti vsak stavek (razen sestavljenega stavka/bloka) zaključen s podpičjem.

3.2.1 Veriga pogojev

Konstrukcija

```
if (pogoj)
    stavek
else if (pogoj)
    stavek
else if (pogoj)
    stavek
else if (pogoj)
    stavek
else
    stavek
```

se pojavi tako pogosto, da zasluži malo bolj podrobnejši opis. Zaporedje stavkov `if` je najbolj splošna pot, kako napišemo odločitev z več možnimi izidi. Vsak stavek je seveda lahko sestavljeni stavek v zavitih oklepajih. Pogoji se testirajo drug za drugim, dokler eden od njih ni izpolnjen. Če noben od pogojev ni izpolnjen, tedaj se izvede stavek za končnim `else`, če obstaja, sicer pa nič.

Vzorec za tako konstrukcijo je funkcija

```
int binsearch(const int x, const int v[], const int n);
```

za binarno iskanje števila `x` po urejeni tabeli `v`, tabela `v` pa ima `n` elementov. Primerjamo iskano vrednost `x` s srednjim elementom v tabeli `v`. Če je `x` manjši od srednjega elementa, ga moramo iskati v levi (spodnji) polovici, sicer pa v desni (zgornji) polovici. Postopek razpolavljanja nadaljujemo, dokler elementa `x` ne najdemo, ali pa ugotovimo, da ga v tabeli ni[1]. Text programa je zapisan v A.12 je na strani 177.

Za lažje programiranje smo v datoteko `"proto.h"` zbrali še vse prototipne deklaracije, potrebne v programih v tem poglavju, glej A.13 na strani 178.

3.3 Stavek switch

Namesto vrste `else if` konstrukcij lahko uporabimo tudi stavek `switch`, če so pogoji oblike "vedno-isti-izraz == konstanten-izraz". Tak stavek ima obliko

```
switch (izraz) {
    case konst-izraz:   stavki
    case konst-izraz:   stavki
    ...
    default:            stavki
}
```

Omejitve na posamezne komponente `switch` stavka najlažje razumemo, če si predstavljamo, kaj prevajalnik s tako konstrukcijo stori. Vsak procesor pozna ukaz za skok (jump, branch). Prevajalnik zbere vse možne konstantne izraze in zgradi tabelo, ki vsebuje skoke na dele programa, ki ustrezajo posameznim primerom. Vrednost izraza `izraz` služi kot indeks za izbiro ustreznega skoka.

---

[1] Pod `UNIX`-om glej `man bsearch`

Konstante v posameznih primerih morajo biti vse različne, pač pa imajo lahko posamezni primeri isti skok v tabeli, torej iste stavke. Kot je opisano, posamezni primeri niso disjunktni ampak se vsak primer nadaljuje (fall through) v naslednjega, če njegovi stavki ne ukažejo drugače. Tipično je zadnji stavek v vsakem primeru stavek `break`, ki pomeni, da je en primer končan in z njim stavek `switch`. Primer `default` je deklica za vse: če se vrednost izraza `izraz` ne ujema z nobenim konstantnim izrazom, tedaj stopi v veljavo primer `default`, če obstaja, sicer pa nič.

V poglavju 1 smo napisali program, ki je štel, koliko znakov posameznih kategorij je na standardnem vhodu. Tam smo uporabili zaporedje `if else` konstrukcij, tu pa bomo uporabili stavek `switch`. Novi izdelek je A.14 na strani 179.

Stavek `break` takoj konča ne samo stavek `switch` ampak tudi vsako zanko, stavek `for`, `do` ali `while`.

Nadaljevanje posameznih primerov v naslednji primer je po svoje koristno, ker lahko pred en primer napišemo več konstant, po drugi strani pa zahteva pozornost, da se to ne zgodi takrat, ko tega ne nameravamo. Zato moramo, razen v primerih kot je bil zgornji (več konstant za en primer), posebej s komentarjem poudariti, da res nameravamo, da se en primer nadaljuje v naslednjega in da to ni pomota.

Stavek `break`, ki sledi zadnjemu primeru, `default` v našem slučaju, formalno ni potreben, vendar ga radi napišemo. Če bomo kdaj dodali še en primer, nam bo takšno defenzivno programiranje morda rešilo kožo.

Naloga 3.1. Naša funkcija `binsearch` ima dva testa v notranjosti zanke, medtem ko bi zadostoval že eden (za ceno več testiranja zunaj zanke). Sestavite verzijo z enim samim testom v zanki in izmerite učinkovitost obeh verzij[2]. Seveda mora biti večkratna vrednost, po kateri iščemo, dovolj velika, pa še ves `binsearch` moramo vdelati v zanko, da nam ne bo merjenje povedalo, da je vse skupaj trajalo nič časa.

Naloga 3.2. Napišite funkcijo

```
void escape(char s[], const char t[])
```

ki prepiše niz `t` v niz `s`, obenem pa znake kot so predelčnik in nova vrsta nadomesti z ubežnimi zaporedji `\t` in `\n`. Uporabite stavek `switch`. Sestavite še funkcijo za pretvorbo v nasprotno smer. Ali lahko obe funkciji testiramo z istim programom, ki v odvisnosti od tega, kako se imenuje, `escape` ali `unescape`, določa, kaj bo delal?

3.4 Zanke

3.4.1 Zanka while

Zanki `for` in `while` smo že videli. V konstrukciji

```
while (izraz)
    stavek
```

---

[2] Pod `UNIX`-om in Kornovo lupino merimo čas izvajanja programa `prog` z ukazom `time prog`

računalnik izračuna vrednost izraza `izraz`. Če je ta vrednost neničelna (true), se stavek `stavek` izvrši, računalnik pa ponovno izračuna vrednost izraza `izraz`. To zaporedje se nadaljuje, dokler vrednost izraza `izraz` ne postane nič (false). Tedaj se izvajanje nadaljuje za `while` stavkom.

3.4.2 Zanka for

Stavek `for` ima obliko

```
for (izraz1; izraz2; izraz3)
    stavek;
```

in je enakovreden konstrukciji

```
izraz1;
while (izraz2) {
    stavek;
    izraz3;
}
```

Ta ekvivalenca se podre, če je v stavku `stavek` prisoten stavek `continue`.

Slovnično so `izraz1`, `izraz2` in `izraz3` izrazi. Običajno so `izraz1` in `izraz3` prirejanja, `izraz2` pa relacija, nujno pa to ni. Vsak od teh treh izrazov lahko manjka, čeprav mora ustrezno podpičje ostati. Če manjkata `izraz1` ali `izraz3`, tedaj pač manjkata tudi v ekvivalentni konstrukciji. Če manjka `izraz2` to pomeni, da je vedno od nič različen (true). Kot poseben primer imamo tako

```
for ( ; ; ) {
    ...
}
```

neskončno zanko, ki jo je treba prekiniti na kak drug način, s stavkom `break` ali `return`.

Odločitev, katero zanko, `for` ali `while`, je bolje uporabiti, je v veliki meri stvar okusa. V situaciji

```
while ((c = getchar()) == ' ' || c == '\n' || c == '\t')
    ;       /* skip white space */
```

je `while` zanka gotovo najbolj naravna. Zanka `for` pa je najbolj primerna v slučaju, da ima zanka enostavno inicializacijo (`izraz1`), enostavno testiranje (`izraz2`) in enostavno popravljanje (`izraz3`), saj zanka `for` združuje vse te elemente na enem mestu. To je najbolj očitno v primeru

```
for (i = 0; i < n; ++i)
```

kar je konstrukcija, s katero lahko obdelamo prvih `n` elementov večkratne vrednosti, ekvivalent stavka `DO` v Fortranu in stavka `for` v Pascalu.

3.4.3 Boljša verzija funkcije myatoi

Sestavimo večji program, recimo boljšo verzijo funkcije `myatoi`, pretvarjanje niza števk v število. Ta verzija bo nekoliko bolj popolna, preskočila bo začetne bele znake in upoštevala predznak. Glej A.15 na strani 180. Standardna glava `<stdlib.h>` vsebuje deklaracijo bolj splošne funkcije, `strtol`[3], ki pretvori niz števk v `long int`.

3.4.4 Shellova metoda sortiranja

Lep primer za uporabo `for` zank nam nudi Shellova metoda sortiranja (D. L. Shell, 1959). Metoda je nekoliko boljša od najbolj zanikrnih metod (direktno vstavljanje ali direktno izbiranje), ki terjajo $O(N^2)$ korakov. Shellova metoda terja $O(N^{1.5})$ korakov. Obstajajo seveda še boljše metode, Quick sort, v povprečju $O(N \times \log(N))$ korakov in, najboljša od vseh, sortiranje s kopicami, garantirano $O(N \times \log(N))$ korakov, a Shellova metoda je vendarle od vseh najbolj preprosta, glej A.16 na strani 181.

3.4.5 Operator vejica

Operator vejica, `,`, pogosto uporabljamo v `for` stavkih. Operator `,` ločuje izraze. Tvorimo lahko izraz

```
izraz1, izraz2, izraz3, izraz4
```

Pomen je: vse izraze, ločene z vejico, računalnik izračuna in vse vrednosti, razen zadnje, zavrže. Vrednost gornjega izraza je torej vrednost izraza `izraz4`. Stavek `for` naravnost kriči po uporabi `,` operatorja, če hočemo kot eno od komponent `for` konstrukcije napisati več kot en izraz. Primer najdemo v funkciji `reverse`, ki obrne niz znakov na mestu, glej A.17 na strani 182.

Tudi jedro zanke `for` smo napisali z operatorjem `,`. Delne izraze, ločene z vejico, pišemo v eni vrsti, da poudarimo, da spadajo skupaj, da tvorijo en sam izraz, ki postane stavek na račun končnega podpičja.

3.4.6 Zanka do

Zanka `while` najprej testira pogoj in če je izpolnjen, izvede jedro in ponovno testira pogoj. Mnogo bolj poredko nam pride prav zanka `do`, ki najprej izvede jedro in potem testira, če bi ga še enkrat. Sintaksa zanke `do` je

```
do
    stavek
while (izraz);
```

Razlika med zanko `do` in zanko `while` je samo na začetku. Zanka `do` zagotovo izvede jedro zanke vsaj enkrat, zanka `while` pa morda niti enkrat ne.

Kot primer uporabe zanke `do` sestavimo funkcijo `itoa`, inverzno funkcijo funkcije `atoi`, glej A.18 na strani 183. Postopek je enostaven. Števke generiramo v napačnem vrstnem redu, zato moramo na koncu niz znakov obrniti s funkcijo `reverse`. Pri tem je zanka `do` skoraj nujna, saj moramo zgenerirati vsaj eno števko, četudi je število nič. Jedro zanke `do` smo vklenili v zavita oklepaja, čeprav je en sam stavek, da ne bi površen bralec razumel končnega `while` kot začetka neke zanke `while`.

---

[3] Pod `UNIX`-om glej `man strtol`

3.5 Stavki break, continue in goto

Včasih se pojavi potreba predčasno prekiniti kakšno zanko. V primeru

```
for ( ; ; ) {
    ...
}
```

je to zagotovo potrebno, saj se sama od sebe taka zanka nikoli ne konča. To nalogo opravi stavek `break`, isti stavek `break`, ki smo ge že uporabljali v stavku `switch`, da se ne bi en primer nadaljeval v naslednjega. Stavek `break` prekine najbolj notranji stavek `switch`, `for`, `while` ali `do`. V primeru zank (in ne v primeru stavka `switch`) lahko uporabimo tudi stavek `continue`, ki povzame naslednjo iteracijo zanke, na primer

```
for (i = 0; i < n; ++i) {
    if (a[i] < 0)           /* skip negative elements */
        continue;
    ...                     /* do positive elements */
}
```

Tipičen primer uporabe stavka `continue` je primer, ko je del zanke, ki sledi, kompliciran in bi inverzija testa predaleč odmaknila konec pogojnega stavka ali pa bi povzročila pregloboko zamikanje.

Primer za stavek `break` je funkcija `trim`, ki zbriše bele znake na koncu niza.

```
#include <ctype.h>
#include <string.h>

int trim(char[]);

/* trim: remove trailing white space; returns new size */
int
trim(char s[])
{
    int     n;

    for (n = strlen(s) - 1; n >= 0; --n)
        if (! isspace(s[n]))
            break;
    s[++n] = '\0';

    return n;
}
```

C pozna tudi (zloglasni) stavek `goto` in oznake, na katere lahko skačemo. Stavku `goto` se vedno lahko izognemo, vendar je včasih najprimernejše orodje in neumno se je pretvarjati, da ne obstaja.

Tipičen primer je `break` iz globine. Stavek `break` funkcionira samo en nivo v globino, če pa hočemo napraviti "večkraten `break`", je to stavek `goto`:

```
for ( ... )
    for ( ... ) {
        ...
        if (disaster)
            goto error;
    }
...
error:
    clean up the mess
```

Oznaka (argument stavka `goto`) ima isto obliko kot ime spremenljivke in temu imenu sledi dvopičje :. Oznaka mora biti definirana v isti funkciji kot stavek `goto`, ki jo uporablja, kar pomeni, da ne moremo skakati iz funkcije v funkcijo, lahko pa "skočimo" na oznako, še preden je definirana, kot na primer v zgornjem primeru.

Še en (upravičen) primer stavka `goto`. Ali imata dve večkratni vrednosti kakšen skupen element?

```
    for (i = 0; i < n; ++i)
        for (j = 0; j < m; ++j)
            if (a[i] == b[j])
                goto found;
    /* did not find any common elements */
    ...
found:
    /* got one: a[i] == b[j] */
    ...
```

Končen sklep ob stavku `goto`. Praviloma ga ne potrebujemo, ni pa treba pretiravati z dogmatizmom.

Naloga 3.3. Sestavite funkcijo

```
void expand(const char s[], char t[])
```

ki prepiše niz `s` v niz `t`. Če je v nizu `s` podniz `"h-k"`, ga funkcija zamenja z nizom `"hijk"`. Bodite pripravljeni na števke, velike in male črke. Kaj naj se zgodi s podnizi `"a-b-c"`, `"a-z0-9"`, `"A-Za-z"`, `"-a-z"`. Vodilni in končni znak - je očitno treba vzeti dobesedno.

Naloga 3.4. V dvojiškem komplementu naša verzija `itoa` ne predela pravilno najbolj negativnega števila. Zakaj ne? Ali znate funkcijo `itoa` popraviti tako, da bo prav tudi v tem primeru?

Naloga 3.5. Napišite funkcijo

```
int itob(const int n, char s[], const int b)
```

ki pretvori število `n` v niz znakov `s` v bazi `b`. Kot poseben primer mora funkcija `itob(n, s, 16)` pretvoriti število `n` v zaporedje znakov v šestnajstiškem sistemu.

Naloga 3.6. Sestavite verzijo funkcije `itoa`, ki sprejme tri argumente namesto dveh. Tretji argument je minimalna širina polja. Konvertirano število je treba na levi dopolniti s presledki, če je število prekratko.

Poglavje 4

Funkcije in struktura programov

4.1 Funkcije

Funkcije razbijejo velik programski posel na več manjših in s tem omogočajo, da ljudje gradimo na temeljih, ki so jih zgradili drugi. Funkcije skrijejo podrobnosti, kako je naloga rešena, če je ostanku programa dovolj, da ve, da je posel opravljen, ne zanima ga pa, kako.

Funkcije v `C` so učinkovite in enostavne za rabo. Tipičen `C` program je sestavljen iz mnogo majhnih funkcij in ne iz nekaj velikih, kot je navada v Fortranu, kjer je pogosto program ena sama velika funkcija. Tekst programa tvorijo posamezne funkcije, ki so lahko zapisane vse na eni sami datoteki ali pa na večih datotekah. Prevajalnik mora prevesti vse sestavne dele, povezovalnik pa jih mora povezati v izvršljivo celoto.

4.1.1 Program mygrep

Sestavimo program, ki bo v nekem tekstu, recimo

```
Ah Love! could you and I with Fate conspire
To grasp this sorry Scheme of Things entire,
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

poiskal vse primerke niza "ould" in izpisal vrstice, ki ta niz vsebujejo, v našem primeru

```
Ah Love! could you and I with Fate conspire
Would not we shatter it to bits -- and then
Re-mould it nearer to the Heart's Desire!
```

To je poenostavitev `UNIX` programa `grep`[1].

Naloga razpade na posamezne dele:

---

[1] Pod `UNIX`-om glej `man grep`

```
while (there is another line)
    if (line contains the pattern)
        print it
```

V luči povedanega lahko sestavimo funkcije, ki opravljajo posamezne podnaloge. Funkcija "`while there is another line`" je funkcija `getline`, ki smo jo napisali v poglavju 1, "`print it`" je funkcija `printf`, ki jo je nekdo že napisal za nas, preostane le preverjanje "`line contains the pattern`".

To preverjanje lahko opravimo s funkcijo `strindex`

```
int strindex(const char source[], const char searchfor[]);
```

ki vrne indeks v `source`, kjer se `searchfor` začne, in `-1`, če niza `searchfor` v nizu `source` ni. Ker se indeksi štejejo od nič naprej, je vrednost `-1` dober indikator, da niza `searchfor` v nizu `source` ni. Če bomo kasneje potrebovali bolj prefinjeno iskanje, bomo samo zamenjali funkcijo `strindex` z bolj prefinjeno, ostanek programa pa je dober. Standardna glava `<string.h>` vsebuje deklaracijo standardne funkcije `strstr`[2], ki je podobna funkciji `strindex`, le da funkcija `strstr` vrne kazalec namesto indeksa.

Kompleten izdelek, program `mygrep`, je program A.19 na strani 184. Funkcija `getline` je malo drugačna kot poglavju 1.

4.1.2 Zgradba funkcij

Definicija vsake funkcije ima obliko

```
tip-rezultata ime-funkcije(deklaracije-argumentov)
{
    definicije, deklaracije in stavki
}
```

Različni koščki lahko manjkajo. Če manjka tip rezultata, prevajalnik privzame, da je `int`. Najbolj oskubljena definicija je

```
dummy() {}
```

funkcija, ki ne stori ničesar in vrne (nedefinirano) celo število (`int`). Take funkcije, pravimo jim štrclji, so koristne med razvijanjem programa, ker omogočajo, da je delno napisan program kljub vsemu slovnično pravilen in ga lahko (brez napak) prevedemo.

4.1.3 Komuniciranje med funkcijami

Program je zaporedje definicij spremenljivk in funkcij. Funkcije komunicirajo med seboj s pomočjo argumentov, s pomočjo vrnjenih vrednosti in eksternih (zunanjih) spremenljivk. Funkcije so lahko napisane na izvorni datoteki v poljubnem vrstem redu, lahko so napisane na več izvornih datotekah pri pogoju, da je vsaka funkcija vsa na eni datoteki.

Stavek `return` je mehanizem, ki poskrbi, da se izvajanje funkcije zaključi in da vrednost funkcije doseže klicatelja funkcije. Sintaksa je

---

[2] Pod `UNIX`-om glej `man strstr`

```
return izraz
```

kjer je `izraz` izraz, ki definira vrednost funkcije. Prevajalnik bo poskrbel za pretvarjanje tipa vrednosti `izraz` v `tip-rezultata`, če je le to mogoče.

Klicajoča funkcija lahko ignorira vrnjeno vrednost, če to želi. Običajno je to napaka, posebej še, če je vrnjena vrednost ignorirana samo včasih. Zato prevajalniki, posebej pa še program `lint`, kričijo, če funkcija vrne vrednost, klicajoča funkcija pa jo ignorira in zato smo skrbno pisali

```
(void)printf(...)
```

in s tem eksplicitno povedali, da nas vrednost funkcije `printf` ne zanima.

Funkcija pa lahko konča izvajanje tudi tako, da se izteče v končni zaviti zaklepaj. V tem primeru je vrednost funkcije nedefinirana, tako kot je nedefinirana, če je v jedru funkcije stavek

```
return;
```

brez izraza `izraz`. V takem primeru je to pravilno le, če je funkcija definirana, da vrne tip `void`, to je, da ne vrne ničesar. Vsekakor je program dvomljiv, če funkcija v nekaterih primerih vrednost vrne, v drugih pa ne. Težava nastane, če klicajoča funkcija pričakuje vrednost, te vrednosti pa (včasih) ni.

4.1.4 Program lint

Standardni program `lint` se obnaša kot prevajalnik, le da ne prevaja ampak le preverja. Programu `lint` moramo navesti vse izvorne datoteke, ker bo pregledal vse in opozoril na vse nedoslednosti in vsa protislovja. Prevajalnik tega ne zmore, ker vidi samo po eno datoteko naenkrat.

4.1.5 Več izvornih datotek

Program `mygrep` lahko napišemo na več datotek, recimo, `main.c`, `getline.c` in `strindex.c`, ki vsaka vsebuje svojo funkcijo. V tem primeru bi prevajalnik aktivirali z ukazom

```
cc main.c getline.c strindex.c -o mygrep
```

Program `cc` bi prevedel datoteke `main.c`, `getline.c` in `strindex.c`, da bi napravil datoteke `main.o`, `getline.o` in `strindex.o`, te datoteke pa bi potem povezovalnik `ld` povezal v izvedljiv program `mygrep`.

4.1.6 Funkcija myatof

Vse naše funkcije doslej so vračale vrednosti tipa `int` ali pa nič, to je, tip `void`. Oglejmo si kako napišemo funkcijo, ki vrne drugačen tip, recimo funkcijo `myatof`, ki pretvori niz znakov v realno število tipa `double`. Funkcija se imenuje `myatof`, ker standardni glavi `<stdlib.h>` in `<math.h>` že vsebujeta deklaracijo ustrezne standardne funkcije `atof`[3].

---

[3] Pod `UNIX`-om glej `man atof`

Funkcija `myatof` zna prežvečiti vodilne bele znake, predznak, celi del, decimalno piko in neceli del, glej A.20 na strani 186. To ni zelo kvalitetna funkcija, njen namen je zgolj ilustracija. Za resno delo uporabljamo funkcijo `atof`.

Tudi klicajoča funkcija mora vedeti, da funkcija `myatof` vrne tip `double`. To lahko povemo s prototipno deklaracijo

```
double myatof(const char []);
```

v *vsaki* funkciji, kjer `myatof` uporabljamo. To delo si lahko olajšamo, če v neko glavo, recimo v `"proto.h"`, vtaknemo vse take deklaracije, potem pa glavo `"proto.h"` vključimo v vse datoteke, ki uporabljajo funkcijo `myatof`. Kot smo že dejali, prototipna deklaracija lahko izpusti imena formalnih parametrov, zadoščajo že njihovi tipi.

4.1.7 Preprosto računalo

Za vzorec sestavimo majhno računalo, komaj dobro za preverjanje izdanih čekov. Program prebere po eno število z vsake vrstice, prišteje to število k tekoči vsoti in izpiše tekočo vsoto. To je program A.21 na strani 187. Deklaracija

```
double  sum;
```

pravi, da je `sum` spremenljivka tipa `double`, deklaracija

```
double  myatof(char []);
```

v glavi `"proto.h"` pa pove, da je `myatof` funkcija, ki vrne tip `double` in ima en argument, tipa `char []`. Funkcija `getline` vrne število prebranih znakov. Ker ima vsaka vrstica vsaj znak `'\n'` na koncu, zahtevamo, da `getline` vrne več kot en znak sicer proglasimo konec računanja.

Če že enkrat imamo funkcijo `myatof`, lahko napišemo funkcijo `myatoi` z njeno pomočjo:

```
/* myatoi: convert string s to integer using myatof */
double  myatof(const char []);

int
myatoi(const char s[])
{
    return (int)myatof(s);
}
```

Funkcija `myatof` vrne vrednost tipa `double`, ki jo moramo pretvoriti v tip `int`. Prevajalnik sicer ve, da je `myatof` tipa `double`, `myatoi` pa tipa `int`, torej bi lahko napravil pretvorbo brez pomoči (brez prisile), vendar prevajalniki včasih ugovarjajo, da pretvorba `double` v `int` zgublja natančnost in prevajalnik lahko zgenerira opozorilo. Temu opozorilu se izognemo s prisilo `(int)`, ki pravi: "Molči, saj vem kaj delam!".

Naloga 4.1. Sestavite funkcijo

```
int strrindex(const char s[], const char t[])
```

ki poišče zadnji (najbolj desni) primerek niza `t` v nizu `s`.

Naloga 4.2. Razširite funkcijo `myatof` tako, da bo razumela tudi desetiški eksponent oblike `e` ali `E`, ki mu sledi (morda) predznačeno število.

4.2 Eksterni objekti

`C` program je množica eksternih objektov, funkcij in spremenljivk. Pridevnik "eksteren" je nasprotje pridevnika "interen". Eksterni objekti so (potencialno) dostopni vsem funkcijam, interni objekti pa so dostopni samo funkciji, kjer so definirani. Funkcije so vedno eksterni objekti, ker `C` ne pozna funkcij znotraj funkcij. Pojma eksteren in interen se torej nanašata na lego objekta. Objekt je interen, če je definiran znotraj neke funkcije, in eksteren, če je definiran zunaj vseh funkcij.

4.2.1 Pomnilniški razredi

Vsak objekt spada v nek (pomnilniški) razred. Takšna razreda sta dva, avtomatičen in statičen razred. Objekti v statičnem razredu imajo stalno dodeljen pomnilnik in ta pomnilnik jim je dodeljen ob povezovanju programa (`ld`). Avtomatični razred živi na skladu. Avtomatični objekti se rodijo ob vstopu v funkcijo in umrejo, to je, izginejo, ko je funkcija končana. Ker se funkcije lahko kličejo med seboj na najbolj čudne načine, na primer rekurzivno, avtomatični objekti ne morejo imeti statično dodeljenega pomnilnika. Funkcije so vedno v statičnem razredu. Eksterne spremenljivke so pravtako v statičnem razredu. Interne spremenljivke pa so lahko v statičnem ali avtomatičnem razredu.

4.2.2 Kategorije

Za vsak objekt je treba povedati še njegovo dostopnost. Objekt je lahko dostopen samo znotraj neke funkcije, znotraj vseh funkcij v neki datoteki ali pa vsepovsod.

Vse te pojme ureja kategorija (storage class). Kategorij je pet:

```
auto, register, extern, static in typedef
```

in vsako od teh petih kategorij lahko napišemo pred vsako deklaracijo. Kategorija `typedef` v resnici sploh ni kategorija v istem smislu kot ostale, le sintaktično je videti kot ostale kategorije in zato velja za kategorijo.

Kategorija `auto` pomeni interni avtomatični razred. Kategorija `register` pomeni interni avtomatični razred skupaj z namigom, da je taka spremenljivka velikokrat uporabljena in naj jo prevajalnik skuša shraniti v enega od "registrov" računalnika, da bo dostop hitrejši. Najprej velja: to je zgolj namig in prevajalnik ima vso pravico, da take namige ignorira, da tretira kategorijo `register` kot kategorijo `auto`. Če pa prevajalnik zmore kakšno spremenljivko res posebej obravnavati, je v njegovi pristojnosti, koliko takih `register` deklaracij bo upošteval in kakšnega tipa smejo biti. Vsekakor, če nekaj ali celo vse `register` deklaracije zamenjamo z `auto` deklaracijami, je program funkcionalno isti kot prej, le nekoliko počasnejši utegne biti. Pa še to ni treba, da je res, saj prevajalnik lahko bolje od nas oceni, katere spremenljivke zaslužijo posebno obravnavo, to je, da si prevajalnik sam izmisli, katere deklaracije naj bodo (za povrh) `register` deklaracije.

Kategorija `extern` pomeni eksterni statični razred. Ker je to eksterni razred, se lahko nanaša na spremenljivke in funkcije. Objekti kategorije `extern` so dostopni od vsepovsod. Tudi kategorija `static` pomeni eksterni statični razred, le da so `static` objekti dostopni le znotraj datoteke, kjer je takšna definicija napisana.

Če kategorije ne predpišemo, so eksterni objekti kategorije `extern`, interni objekti pa kategorije `auto`. Za eksterne objekte je torej smiselno le, da predpišemo kategorijo `static`, za interne objekte pa `register`, `static` ali `extern`.

Eksternim spremenljivkam pogosto pravimo globalne spremenljivke, ker jih lahko dosegajo vse funkcije. Globalne spremenljivke so alternativa za funkcijske parametre. Na videz so bolj udobne kot parametri: globalna spremenljivka je pač tam, kjer jo želimo. Težava je v tem, da je tam tudi, če je ne želimo. Program, ki uporablja globalne spremenljivke, ima preveč povezav s "svojim rojstnim krajem". Globalnim spremenljivkam se izogibamo. V skrajni sili uporabljamo `static` spremenljivke. Tudi funkcije definiramo kot `static`, če jih ne potrebujemo izven datoteke, kjer so definirane.

4.2.3 Računalo

Sestavimo malo večji primer in sicer računalo. Ker bo tako lažje, bomo sestavili računalo, ki uporablja obratno Poljsko notacijo, kot jo uporabljajo Forth, Postscript in nekatera žepna računala.

Standardna notacija je infiksna: operator je napisan med obema operandoma. Pri obratni Poljski notaciji operator napišemo za (vsemi) operandi. Infiksni izraz

```
(1 - 2) * (4 + 5)
```

v obratni Poljski notaciji napišemo kot

```
1 2 - 4 5 + *
```

V obratni Poljski notaciji ne potrebujemo oklepajev, vse dokler uporabljamo operatorje z znanim številom operandov.

Celo izvedba je preprosta. Vsak operand, na katerega naletimo, potisnemo na sklad. Ko naletimo na operator, s sklada poberemo znano število argumentov, uporabimo na njih operator, in potisnemo rezultat nazaj na sklad.

Skelet programa je dovolj preprost:

```
while (next operator or operand is not end-of-file indicator)
    if (number)
        push it
    else if (operator)
        pop operands
        do operation
        push result
    else if (newline)
        pop and print top of stack
    else
        error
```

Operaciji `push` in `pop` sta trivialni, a ko dodamo še vse potrebne kontrole, je bolje, da sta to dve posebni funkciji. Pravtako potrebujemo posebno funkcijo, ki bo s standardnega vhoda prebrala naslednji operator ali naslednji operand.

Nerešeno je še vprašanje, kje je sklad in katere funkcije imajo dostop do njega. Ena možnost je, da sklad sedi v funkciji `main` in da ga kot parameter posredujemo funkcijama `push` in `pop`. Vendar funkciji `main` ni treba vedeti za spremenljivke, ki kontrolirajo sklad, `main` samo kliče `push` in `pop`. Zato bomo spremenljivke, ki jih potrebujeta `push` in `pop` definirali šele tik pred `push` in `pop` funkcijama. Dokler prevajalnik ne vidi definicije ali deklaracije funkcije, le ta namreč ne obstaja.

Tako dobimo naslednji skelet:

```
#include's
#define's

prototype declarations

function declaration for main

int main(void) { ... }

external variables for push and pop

void push(double f) { ... }
double pop(void) { ... }

int getop(char s[]) { ... }

routines called by getop
```

Kasneje bomo ta skelet razbili na več izvornih datotek.

Program `calc` (program A.22) je na strani 188. Ker sta + in * komutativna operatorja, je vrstni red operandov brezpredmeten, v stavku

```
push(pop() + pop());
```

nam je vseeno, ali bo prevajalnik sestavil kodo tako, da bo najprej poklical funkcijo `pop` pred znakom `+` ali pa tisto za znakom `+`. Pri operatorjih `-` in `/` pa temu ni tako. Na vrhu sklada imamo minuend (divizor) in šele pod njim subtrahend (dividend), zato moramo z vrha sklada potegniti minuend (divizor) v `op2`, šele nato lahko napišemo

```
push(pop() - op2);
```

oziroma

```
push(pop() / op2);
```

Funkciji `push` in `pop` sta spet preprosti. Kot smo že rekli, sklad `val` in indeks na sklad `sp` sta potrebna samo funkcijama `push` in `pop`, ne moreta pa biti lokalni spremenljivki v eni od obeh funkcij, ker ju mora dosegati tudi druga funkcija.

V nadaljevanju imamo funkcijo `getop`. Le ta najprej pomeče proč vodilne bele znake, nato pa zbere vse znake, ki tvorijo število, to je desetiške števke (morda s piko) v večkratno vrednost `s`. Funkcija vrne operator (kot znak, recimo, `+`) ali pa `NUMBER`, če hoče povedati, da je videla število.

Preostaneta nam še funkciji `getch` in `ungetch`. Med branjem znakov beremo znake, dokler so del števila. Ko naletimo na znak, ki ni več del števila, vemo, da je števila konec, prebrali smo pa en znak preveč. V Pascalu se s tem problemom otepamo tako, da Pascal dopušča, da pokukamo na standardni vhod, kakšen je naslednji znak, ne da bi ga prebrali. V `C` tega ne moremo, zato znak preberemo in če ga ne maramo, ga "odpreberemo", pošljemo ga nazaj. To seveda ne gre dobesedno tako. Zato bo funkcija `ungetch` shranila prebrani znak na nek sklad, funkcija `getch` pa bo najprej pogledala na sklad in pobrala znak s sklada, če je tam, sicer ga bo pa res prebrala s standardnega vhoda.

V standardni glavi `<stdio.h>` je deklaracija funkcije `ungetc`, ki zmore vrniti samo en prebran znak naenkrat. Naša verzija zna hkrati vrniti `BUFSIZE` znakov, zato pa je standardni `ungetc` bolje integriran v `<stdio.h>`. Vse funkcije, ki berejo, se zavedajo, da imajo lahko en znak vrnjen nazaj. Za znake, vrnjene z `ungetch`, pa ve samo `getch`.

4.2.4 Program na več izvornih datotekah

Funkcije in spremenljivke, ki tvorijo program, ni treba, da sede v eni sami datoteki. Pri povezovanju programa se lahko sklicujemo še na že prevedene funkcije, ki so shranjene v kakšni knjižnici. Ob tem se pojavijo naslednja vprašanja:

- Kako napišemo deklaracije, da bodo spremenljivke med prevajanjem pravilno deklarirane?
- Kako organiziramo deklaracije, da se med povezovanjem kosi prav staknejo?
- Kako organiziramo deklaracije, da bo obstajala ena sama kopija?
- Kako eksternim spremenljivkam predpišemo začetne vrednosti?

Na ta vprašanja bomo odgovorili na primeru programa `calc`. To je seveda premajhen program, da bi bilo rezanje na več datotek upravičeno, vendar je to dobra ilustracija naše problematike.

4.2.5 Doseg imen

Doseg imena je tisti del programa, kjer ime lahko uporabljamo. Doseg avtomatičnih spremenljivk, definiranih na začetku funkcije, je vsa funkcija. Takim imenom pravimo lokalna imena. Lokalna imena v eni funkciji nimajo nobene zveze z lokalnimi imeni v drugi funkciji. Isto velja za parametre funkcije, ki so v bistvu lokalne spremenljivke.

Doseg imena eksterne spremenljivke ali funkcije sega od točke, kjer je ime deklarirano, do konca datoteke. Če so imena deklarirana takole:

```
int main(void) { ... };

static int sp = 0;
static double val[MAXVAL];

static void push(double f) { ... }
static double pop (void) { ... }
```

lahko imena `sp` in `val` uporabljamo v funkcijah `push` in `pop` tako, da jih enostavno navedemo. Ti dve imeni in imeni `push` in `pop` pa v funkciji `main` niso vidna (brez dodatnih predhodnih deklaracij).

Če neko ime uporabimo v neki datoteki še preden ga definiramo, ali če je ime definirano v neki drugi datoteki, tedaj moramo v prvi datoteki (pred uporabo imena) napisati `extern` deklaracijo. Deklaracije

```
extern int sp;
extern double val[];
```

povedo (preostanku datoteke), da sta `sp` in `val` nekje definirana in povezovalnik bo že znal povezati ti dve deklaraciji z dejanskima definicijama

```
static int sp = 0;
static double val[MAXVAL];
```

Včasih je težko na pogled ločiti med definicijo in deklaracijo. Definicija definira tip količine in lahko pove njeno začetno vrednost. Deklaracija po drugi strani pove samo tip. Če je prisotna začetna vrednost, je to gotovo definicija. Če je kot definicija nepopolna, recimo manjka število komponent v večkratni vrednosti, tedaj je to samo deklaracija. Kaj pa je

```
int xyz;
```

Lahko je definicija ali deklaracija. Prvi tak primerek je definicija, vsi ostali so deklaracije.

Razdelimo definicije v dve datoteki, `file1` in `file2`, takole:

```
file1:

    extern int sp;
    extern double val[];

    void push(double f) { ... }

    double pop(void) { ... }

file2:

    int sp = 0;
    double val[MAXVAL];
```

To ni zelo verjetna razdelitev za program `calc`, služi samo za ilustracijo principa. V datoteki `file1` imamo deklaraciji količin `sp` in `val` ter definiciji količin `push` in `pop`, v datoteki `file2` pa imamo samo definiciji količin `sp` in `val`.

4.2.6 Program calc

Oglejmo si sedaj, kako razdelimo program `calc` v več datotek. Ves program je sicer premajhen, da bi se to res splačalo, zato je treba to razdelitev spet razumeti kot ilustracijo ideje, ki je koristna v večjem programu. Tekst programa bomo razdelili na štiri datoteke:

- `main.c` s funkcijo `main`,
- `getop.c` s funkcijo `getop`,
- `stack.c` s skladom, funkcijo `push` in funkcijo `pop`,
- `getch.c` s funkcijama `getch` in `ungetch`.

Preostane vprašanje, kje so deklaracije. Idealno bi bilo, da sestavimo nekaj glav, ki vsebujejo deklaracije, potrebne več kot v eni datoteki. Ampak tako bi se nabralo preveč glav, zato, pri razumno velikem programu, zberemo vse glave v eno in jo vključimo povsod, kjer kaj iz nje potrebujemo.

Tako pristanemo pri naslednji organizaciji:

```
                   calc.h
                   ------
                   #define NUMBER '0'
                   void push(double);
                   double pop(void);
                   int getop(char []);
                   int getch(void);
                   void ungetch(int);

main.c:            getop.c:           stack.c:
-------            --------           --------
#include <stdio.h> #include <stdio.h> #include <stdio.h>
#include <math.h>  #include <ctype.h> #include "calc.h"
#include "calc.h"  #include "calc.h"  #define MAXVAL 100
#define MAXOP 100  int getop() {      static int sp = 0;
int main () {         ...             static double val[MAXVAL];
   ...             }                  void push (double f) {
}                                         ...
                                      }
                                      double pop(void) {
                   getch.c                ...
                   -------            }
                   #include <stdio.h>
                   #define BUFSIZE 100
                   static char buf[BUFSIZE];
                   static int bufp = 0;
                   int getch(void) {
                       ...
                   }
                   void ungetch(int) {
                       ...
                   }
```

V novem programu smo pridno uporabljali statične deklaracije, na primer `sp` in `val` v `stack.c` ter `buf` in `bufp` v `getch.c`. Zamisel je, da skrijemo objekte, spremenljivke in funkcije, ki jih uporabljamo samo v eni datoteki, pred programskim tekstom v drugih datotekah. V vohunskih in vojaških krogih pravijo temu principu: "Need to know", vsakdo naj vé samo to, kar mora nujno vedeti.

4.2.7 Statične interne spremenljivke

Pridevnik `static` lahko uporabljamo tudi za interne spremenljivke, spremenljivke, definirane znotraj neke funkcije. Taka spremenljivka je še vedno vidna samo v funkciji, kjer je definirana, a sedi v statičnem pomnilniku, kot eksterne spremenljivke, zato preživi konec funkcije. Izven funkcije spremenljivka seveda ni dostopna, pač pa postane dostopna, če se vrnemo v funkcijo, kjer je definirana, in pri tem je njena vrednost taka, kot je bila, ko je program funkcijo zapustil. Kot kontrast, avtomatične spremenljivke se zgrade vsakokrat na novo in ne ohranijo starih vrednosti.

4.2.8 Register spremenljivke

Pridevnik `register` nakazuje prevajalniku, da bomo spremenljivko (ali parameter) na veliko uporabljali in naj jo prevajalnik shrani v hitreje dostopen pomnilnik, če le tak obstaja. Take deklaracije lahko uporabljamo le za avtomatične spremenljivke in formalne parametre. Prevajalnik ima vso pravico, da take nasvete ignorira. Primer:

```
void f(register unsigned m, register long n)
{
    register int    i;

    ...
}
```

Prevajalnik se na lastno pest odloči, koliko deklaracij `register` lahko upošteva, in kakšne. Odvečne deklaracije `register` se vedejo, kot da pridevnika `register` ni. Obstaja formalna omejitev, da ne smemo napisati "naslova" registrske spremenljivke, neodvisno od tega, ali je prevajalnik nasvet ubogal ali ne. Na kratko, deklaracije `register` so zgodovina (iz ne tako davnih časov), ko smo imeli zanikrne prevajalnike, ki niso znali sami razporediti avtomatičnih spremenljivk po registrih.

4.2.9 Bločna struktura

`C` ni bločno strukturiran jezik v smislu Pascala, ker ne moremo definirati funkcij znotraj funkcij. V vseh ostalih ozirih pa `C` je bločno strukturiran jezik. Specialno, vsak sestavljeni stavek ima lahko svoje definicije in deklaracije, ki jih lahko uporabljamo, da skrijemo spremenljivke pred ostalimi deli funkcije. Sestavljeni stavek, ki ima svoje definicije ali deklaracije, se imenuje blok. Ob tem skrivanju lahko nova definicija ali deklaracija istega imena zahteva, da pomeni sedaj ime naj drugega.

Primer.

```
if (n > 0) {
    int     i;      /* novi "i", velja do konca bloka */

    for (i = 0; i < n; ++i)
        ...
}                   /* konec veljavnosti novega "i" */
```

Avtomatično spremenljivko, definirano in inicializirano v notranjem bloku, program vsakokrat ponovno zgradi in inicializira. Statična spremenljivka, definirana in inicializirana v notranjem bloku, je še vedno inicializirana samo enkrat, ob začetku programa, bolje rečeno, med povezovanjem programa.

Avtomatične spremenljivke, vključno s formalnimi parametri, zakrijejo globalne spremenljivke in funkcije z istimi imeni. V primeru

```
int     x;
int     y;

void f(double x)
{
    double  y;

    ...
}
```

je v jedru funkcije `x` formalni parameter, ki je tipa `double`, zunaj jedra pa je `x` globalna spremenljivka tipa `int`. Podobno velja za `y`. Takšno skrivanje obstoječih imen je idealno sredstvo za sestavljanje ugank tipa: "Kaj stori naslednji program?", pri resnem programiranju pa se temu izogibamo, saj je napisane programe že brez tega dovolj težko razumeti.

4.2.10 Inicializacija

Inicializacijo smo že večkrat omenjali. Ob definiciji spremenljivke lahko povemo tudi njeno začetno vrednost. To lahko storimo tudi, če definicija pravi, da je vrednost spremenljivke konstantna (`const`), in v tem primeru je inicializacija edini način, da vrednost spremenljivke sploh definiramo. Začetne vrednosti neinicializiranih spremenljivk so:

- vsi biti nič v primeru statičnih spremenljivk,
- nedefinirane v primeru avtomatičnih spremenljivk.

Specialno to pomeni, da je naslednja funkcija trapasta:

```
int traparija(void)
{
    int     x;

    return x >= 0 ? +1 : -1;
}
```

ker je vrednost funkcije prepuščena naključju, a nikar zato ne mislite, da bi lahko služila kot generator slučajnih števil.

Začetne vrednosti statičnih spremenljivk morajo biti konstantne, torej znane med prevajanjem, začetne vrednosti avtomatičnih spremenljivk pa so poljubni izrazi.

4.2.11 Rekurzija

Funkcije v jeziku `C` so lahko rekurzivne. Rekurzivne funkcije so pogosto najpreprostejša rešitev sicer kompliciranega problema, recimo

```
/* printd: print n in decimal */
void printd(int n)
{
    if (n < 0) {
        (void)putchar('-');
        n = -n;
    }
    if (n / 10)
        printd(n / 10);
    (void)putchar(n % 10 + '0');
}
```

Rekurzivna rešitev problema ne bo prihranila nič pomnilnika, ker mora sklad, kamor se shranjujejo vrednosti, kljub vsemu nekje biti. Prav tako računalnik ne bo prihranil na času. Pač pa je rekurziven program pogosto lažje razumeti, lažje je samega sebe prepričati, da res dela tisto, kar obljublja.

Zelo dober primer rekurzije je `quicksort`, metoda za urejanje tabel, ki jo je leta 1962 izumil C.A.R. Hoare. V dani večkratni vrednosti izberemo en element, preostale pa razdelimo v dve podmnožici. V eno podmnožico potaknemo elemente, ki so manjši od izbranega elementa, preostale elemente pa v drugo podmnožico. Isti postopek ponovimo sedaj na vsaki podmnožici. Ko ima podmnožica manj kot dva elementa, je ni treba več deliti naprej in ta pogoj ustavi rekurzijo. Glej program A.23 na strani 191.

Rekurzija pa ni vedno priporočljivo orodje. Program, ki ga brez težav napišemo z zanko, je pogosto slab primer za rekurzijo, čeprav je matematična formulacija rekurzivna.

Šolski primer napačne uporabe rekurzije je Fibonaccijevo zaporedje. Sestavimo program, ki natiska n-to Fibonaccijevo število, če velja

$F_{n+1} = F_n + F_{n-1}$

$F_0 = F_1 = 1$

Program lahko napišemo z zanko ali pa z rekurzijo, glej program A.25 na strani 193. Porabljen čas za rekurzivno verzijo v sekundah (100 MHz Pentium) je

n	*time*
34	$3.23s$
35	$5.20s$
36	$8.40s$
37	$13.59s$
38	$21.98s$

in hitro vidimo, da časi naraščajo pravtako kot Fibonaccijevo zaporedje. Časi zančne verzije so komajda merljivi:

n	*time*
10^3	$0.01s$
10^4	$0.01s$
10^5	$0.01s$
10^6	$0.03s$
10^7	$0.25s$
10^8	$2.28s$

Seveda so v zančni verziji členi Fibonaccijevega zaporedja že zdavnaj prekoračili obseg celih števil, a na zapravljen čas to ne vpliva (v bistvu je to modularna aritmetika).

Nauk je preprost: če se da napisati zanko, je rekurzija neumnost.

4.3 Predprocesor

K `C` jeziku sodi še `C` predprocesor. To je program, ki prepiše napisan program v dejanski `C` program, ki ga potem pravi prevajalnik prevede. Ta opis ni treba, da se ujema z resnico: predprocesor je lahko kar spojen s pravim prevajalnikom v eno celoto. Vsekakor pa si bolj zanesljivo pravilno predstavljamo, kaj se dogaja, če privzamemo, da je predprocesor ločen program.

4.3.1 Direktiva #include

Transformacije, ki jih dela predprocesor, so v prvi vrsti `#include`. Vsako vrstico oblike

```
#include "filename"
```

ali

```
#include <filename>
```

predprocesor zamenja z vsebino datoteke `filename`. V obliki `"filename"` prevajalnik najprej poskuša najti datoteko tam, kjer je našel izvorno datoteko, če je pa tam ni, ali pa če uporabimo `<filename>` obliko, pa "na standardnem mestu[4]". Direktive `#include` lahko gnezdimo.

4.3.2 Direktiva #define

Drug pomemben primer uporabe predprocesorja je substitucija makrojev. Definicija makroja ima obliko

```
#define ime     nadomestni tekst
```

in pove, da je treba besedo `ime` povsod zamenjati z nadomestnim tekstom. Ime `ime` se mora začeti s črko (podčrtaj velja za črko), sledijo pa dodatne črke in števke. Imena makrojev naj bodo tvorjena brez malih črk - to jih identificira kot makroje. Imena makrojev, ki se začenjajo s podčrtajem, so nekako rezervirana za standardno knjižnico in za taka imena je bolje, da jih ne uporabljamo.

---

[4] Pod (`SVR4`) `UNIX`-om je to standardno mesto imenik `/usr/include`

Nadomestni tekst je poljuben tekst, formalno v eni vrstici. Tej formalni omejitvi se izognemo, če vrstico končamo z nagibnico. Nagibnica, ki ji sledi znak `'\n'` skupaj z znakom `'\n'` izgine in prevajalnik bo videl eno samo (dolgo) vrsto. Ta trik velja povsod v izvorni datoteki, čeprav drugje ni tako koristen.

Predprocesor se mora odločiti, kje bo prepoznaval makroje. Denimo, da imamo v programu definicijo

```
#define WALK poljuben tekst
```

V nadaljevanju imamo tekst

```
x = y + WALK / 12;
x = y+WALK/12;
WALK = "WALK";
WALK = WALKMAN;
```

V vseh štirih vrsticah bo predprocesor nadomestil makro `WALK` s `poljuben tekst`, v zadnjih dveh vrsticah samo po enkrat: v narekovajih predprocesor makroja ne bo prepoznal, niti ne makroja, ki mu takoj sledi `MAN`. Konstrukcija `WALKMAN` nima z makrojem `WALK` nobene zveze.

Nadomestni tekst makroja je poljuben. Programer je odgovoren, da je po zamenjavi imena z nadomestnim tekstom končni izdelek res legalen `C` program (, če gre res za `C` program). Naslednje je formalno pravilna, a ne priporočljiva uporaba predprocesorja:

```
#define forever for ( ; ; )

forever {
    ...
}
```

je na skrit način zapisana neskončna zanka.

Bolj zanimivi so makroji s parametri. Če v definiciji makroja imenu makroja takoj sledi predklepaj, bo to makro s parametri. (Formalni) parametri makroja so spet imena, tvorjena po ustaljenih načelih. Formalni parametri, če jih je več, morajo biti ločeni z vejico, za zadnjim (ali edinim) parametrom pa sledi zaklepaj, nekaj belega prostora, in nadomestni tekst. V nadomestnem tekstu imamo spet lahko formalne parametre, ki se pri nadomeščanju zamenjajo z dejanskimi parametri makroja. Primer:

```
#define max(A, B)   ((A) > (B) ? (A) : (B))
```

definira makro z imenom `max`, ki ima dva formalna parametra, `A` in `B`. V nadaljevanju lahko sedaj pišemo

```
x = max(p + q, r + s);
```

in predprocesor bo to vrstico nadomestil z

```
x = ((p + q) > (r + s) ? (p + q) : (r + s));
```

torej res dobimo večjega od obeh izrazov. V makrojih moramo zelo previdno uporabljati oklepaje: vsak formalni parameter v nadomestnem tekstu denemo v oklepaje, prav tako tudi kompleten nadomestni tekst. S tem se izognemo neljubim presenečenjem. Konstrukcijo

```
#define square(x)   x * x
z = square(z + 1);
```

bo predprocesor nadomestil z

```
z = z + 1 * z + 1;
```

to pa ni tisto, kar smo nameravali. Podobno bo iz

```
#define sum(x, y)   x + y
z = sum(i, j) * sum(x, y)
```

predprocesor napravil

```
z = i + j * x + y;
```

V teh primerih se tragediji izognemo z oklepaji, v primeru

```
x = max(i++, j++);
```

pa, seveda, dobimo

```
x = ((i++) > (j++) ? (i++) : (j++));
```

torej se večji od `i` in `j` poveča dvakrat. To je nekaj česar z makroji pač ne počnemo. Posebno stranski učinki operatorjev povečevanja (zmanjševanja) so zelo hudobni.

Kljub navedenim nevarnostim so makroji nadvse koristni. Standardna funkcija `getchar` je pogosto izvedena kot makro, da bi se izognili (sicer majhni) zamudi pri klicanju nove funkcije. Če pa želimo biti prepričani, da je `getchar` funkcija in ne makro, tedaj lahko napišemo

```
#undef getchar
int getchar(void) { ... }
```

in bo predprocesor "pozabil" makro `getchar`, če je kdaj sploh vedel zanj, torej bo `getchar` zagotovo funkcija (, ali pa neznano ime).

Rekli smo že, da predprocesor v narekovajih ne prepoznava imen makrojev. To velja tudi za formalne parametre makroja. Če pa je pred formalnim parametrom, recimo `expr`, znak `#`, bo predprocesor konstrukcijo `#expr` nadomestil z "`dejanski-parameter`", to je z dejanskim parametrom v narekovajih. Tako kostrukcijo uporabljamo na primer takole:

```
#define dprintf(expr)   (void)printf(#expr " = %g\n", expr)

dprintf(x/y);
```

Predprocesor bo zadnjo vrstico nadomestil z

```
(void)printf("x/y" " = %g\n", x/y);
```

in ker kot zadnji korak predpocesor stakne zaporedne nize v en sam niz, dobimo

```
(void)printf("x/y = %g\n", x/y);
```

Preden bo začel predprocesor s to telovadbo, bo v dejanskem parametru vsak znak " nadomestil z \" in vsak \ z \\, zato da bo končni izraz legalen niz.

Predprocesor ima še en tovrsten operator, `##`, s katerim lahko med substitucijo makroja staknemo dva dejanska parametra. Potem ko bo predprocesor odstranil znaka `##` in prazen prostor okrog njiju, bo predprocesor vrstico še enkrat pregledal, če je sedaj morda v njej še kakšen makro, ki ga prej ni videl. Primer za uporabo tega operatorja je makro

```
#define paste(front, back)  front ## back
```

Na osnovi te definicije bo

```
paste(name, 1)
```

postalo ime "name1".

4.3.3 Direktiva #if

Pogojno vključevanje teksta je naslednja funkcija predprocesorja, sicer močno zlorabljan mehanizem. Del teksta lahko pogojno vključimo v prevajanje ali pa ne. Direktiva

```
#if konstanten-izraz
```

pravi, da mora predprocesor izračunati `konstanten-izraz`. Če je njegova vrednost od nič različna (true), tedaj bo predprocesor tekst do direktive `#else`, `#elif` ali `#endif` vključil v prevajanje, v nasprotnem primeru pa ne. Konstanten izraz bo izračunal predprocesor, ki o `C` ne ve kaj prida, zato konstanten izraz ne sme vsebovati `sizeof` operatorja, prisil in enumeracijskih konstant. Kot rečeno, vključen (ali izključen) tekst sega do `#elif` direktive, `#else` ali `#endif` direktive. Pri `#else` direktivi se smisel testa obrne: če je bil tekst doslej vključen, od sedaj naprej ni več, če do sedaj ni bil, od sedaj naprej je. Direktiva `#elif` pomeni else if, torej je to `#else` z dodatnim pogojem. Cela kača sa konča z direktivo `#endif`.

Primer take kače je testiranje simbola `SYSTEM`. Na nek način definiramo simbol `SYSTEM` kot eno od prepoznavnih imen `SYSV`, `BSD`, `MSDOS` in za vsako od teh imen imamo drugačno glavo `HDR`:

```
#if SYSTEM == SYSV
    #define HDR "sysv.h"
#elif SYSTEM == BSD
    #define HDR "bsd.h"
#elif SYSTEM == MSDOS
    #define HDR "msdos.h"
#else
    #define HDR "default.h"
#endif

#include HDR
```

Pogosto v `#if` pogoju uporabljamo (edino) funkcijo, ki jo predprocesor razume, `defined`. Konstanten izraz `defined(ime)` ima od nič različno vrednost (true), če je ime `ime` v predprocesorju definirano, kot na primer

```
#define ime        /* sedaj je ime definirano,
                    * defined(ime) je res
                    */

#undef  ime        /* predprocesor je pravkar pozabil
                    * ime, ni res defined(ime)
                    */
```

Pogosto uporabljamo poenostavitev

```
#ifdef ime         /* isto kot #if defined(ime) */
#ifndef ime        /* isto kot #if !defined(ime) */
```

Predprocesorjevo pogojno vključevanje teksta s pridom uporabljamo v (standardnih) glavah, da bi preprečili nenamerno večkratno vključevanje iste glave. Denimo, da imamo glavo `"hdr.h"` in nas skrbi, da bo kdo večkrat vključil `"hdr.h"`, to bi bilo pa napak, kdo ve zakaj. Večkratno vključevanje lahko preprečimo z naslednjim prijemom

```
#if !defined(HDR_H)
#define HDR_H
/* normalna vsebina hdr.h */
#endif
```

Prvič, ko vključimo `"hdr.h"`, simbol `HDR_H` še ne bo definiran, zato bo pogoj izpolnjen in predprocesor bo zato definiral simbol `HDR_H` in vključil preostanek teksta. Pri kasnejših vključitvah bo predprocesor datoteko `"hdr.h"` v bistvu preskočil. Simbol, ki ga uporabljamo, je `HDR_H`, ne `HDR.H`, saj to ni en simbol.

Naloga 4.3. Program `calc` se da preprosto razširiti. Dodajte operator ostanka `%` in negativna števila.

Naloga 4.4. Dodajte ukaz, ki

- izpiše vrhnji element sklada ne da bi ga zbrisal;
- podvoji element na vrhu sklada, to je, napravi še eno kopijo;
- zamenja vrhnja dva elementa na skladu;
- sprazni sklad.

Naloga 4.5. Dodajte dostop do standardnih funkcij kot so `sin`, `exp` in `pow`.

Naloga 4.6. Dodajte ukaze za spremenljivke. Naloga je preprosta, če je spremenljivk 26, po ena za vsako črko abecede. Dodajte še spremenljivko za vrednost zadnjega izpisanega števila.

Naloga 4.7. Sestavite funkcijo

```
void ungets(char s[])
```

ki bo vrnila cel niz. Ali naj `ungets` kaj ve o `buf` in `bufp` ali naj kar uporablja `getch`, `ungetch`?

Naloga 4.8. Denimo, da nikoli ne bo treba vrniti več kot en znak naenkrat. Ustrezno modificirajte `getch` in `ungetch`.

Naloga 4.9. Naši funkciji `getch` in `ungetch` ne obravnavata pravilno vrednosti `EOF`. Premislite, kako naj se obnašata, če vrnemo `EOF` in ta načrt uresničite.

Naloga 4.10. Alternativna organizacija uporablja `getline`, da prebere celo vrstico naenkrat. S to organizacijo postaneta `getch` in `ungetch` nepotrebna. Modificirajte program `calc` v tej smeri.

Naloga 4.11. Adaptirajte ideje funkcije `printd` na funkcijo `myitoa`, ki shrani decimalno reprezentacijo števila v niz.

Naloga 4.12. Sestavite rekurzivno verzijo funkcije `reverse`, ki obrne niz znakov na mestu.

Naloga 4.13. Definirajte makro `swap(t,x,y)`, ki zamenja dva parametra, `x` in `y` tipa `t` med seboj. Bločna konstrukcija Vam bo pri tem močno koristila.

Poglavje 5

Kazalci in večkratne vrednosti

5.1 Kazalci

Kazalec je spremenljivka, katere vrednost je naslov neke druge spremenljivke. Kazalce v `C` na veliko uporabljamo, deloma zato, ker so često edini način, da izrazimo nek račun, deloma zato, ker pogosto vodijo do krajšega in učinkovitejšega koda. Kazalci in večkratne vrednosti so tesno povezani in to povezavo bomo v tem poglavju vzeli pod mikroskop.

Kazalce pogosto obtožujejo, da predstavljajo, skupaj z `goto` stavkom, čudovit mehanizem, kako skonstruirati nerazumljive programe. To zagotovo drži, ni pa krivo orodje, kriv je obrtnik. Zlahka se nam zgodi, da kazalec kaže na nepričakovano mesto, posledice pa so napačen program. Temu se je treba seveda izogibati in izogibati se dá, le vedeti je treba, kaj počnemo[1].

Rekli smo, da je v ozadju modela računalnika, ki ga uporablja `C`, pomnilnik, ki ga tvorijo zlogi. Vsak zlog pomnilnika ima svoj naslov. Ta naslov je, tipično, dvaintridesetbitno število, tako kot `long`. Če hočemo početi čudne reči (raje ne!), tedaj lahko `long` vrednost zapišemo v pomnilnik na mesto, kjer je shranjen nek kazalec, ali pa kazalec prepišemo v `long` spremenljivko. Prevajalnik se bo temu sicer skušal nekoliko upirati, lahko ga pa prepričamo, da "jaz že vem, kaj delam".

5.1.1 Adresni operator &

Obstaja popolnoma legalna pot za manipulacije s kazalci. Če je `c` nek znak, `char`, in `p` kazalec, ki kaže na ta znak, lahko to dosežemo s prirejanjem

```
p = &c;
```

Unarni operator `&` je adresni operator, njegova vrednost je "naslov" njegovega operanda. Adresni operator lahko uporabljamo samo na objektih v pomnilniku, na spremenljivkah in na komponentah večkratne vrednosti, ne smemo ga pa uporabiti na izrazih, konstantah in na `register` spremenljivkah.

---

[1] V takem slučaju je orodje z imenom `PURIFY` čudovit pripomoček

5.1.2 Operator dereferenciranja *

Unarni operator `*` je operator dereferenciranja. Če ga uporabimo na kazalcu, dobimo vrednost, na katero kazalec kaže. Denimo, da sta `x` in `y` spremenljivki tipa `int`, `ip` pa kazalec na `int`. Iz naslednjega (za lase privlečenega) zaporedja lahko razberemo, kako definiramo kazalce in kako uporabljamo operatorja `*` in `&`.

```
int     x = 1, y = 2, z[10];
int     *ip;                /* ip is a pointer to int */

ip = &x;                    /* ip now points to x */
y = *ip;                    /* y is now 1 */
*ip = 0;                    /* x is now 0 */
ip = &z[0];                 /* ip now points to z[0] */
```

Definicije spremenljivk `x`, `y` in `z` niso nič novega. Definicija kazalca `ip`

```
int     *ip;
```

je namenjena kot mnemotehničen pripomoček: pravi, da je `*ip` nekakšen `int`. Definicija in deklaracija kazalca posnemata sintakso izraza, kjer utegnemo tak kazalec uporabiti. Isti razmislek velja za deklaracije funkcij, na primer

```
double  *dp, atof(char *);
```

pove, da sta izraza `*dp` in `atof(s)` tipa `double` in da je argument funkcije `atof` kazalec na `char`.

Najbrž boste opazili implikacijo, da je vsak kazalec omejen, da kaže vedno na objekte istega tipa. Izjema je `void *`, "kazalec na void", ki lahko hrani kakršenkoli kazalec, ne moremo ga pa dereferencirati (dereferencirati v kaj?).

Če `ip` kaže na število `x`, tedaj lahko uporabimo `*ip` v vsakem kontekstu, kjer lahko uporabimo `x`, na primer

```
*ip = *ip + 10;
```

poveča `*ip` za deset.

Unarna opratorja `*` in `&` vežeta tesneje kot aritmetični operatorji, tako da prirejanje

```
y = *ip + 1;
```

vzame to, na kar `ip` kaže, prišteje ena in shrani vsoto v `y` in

```
*ip += 1
```

poveča količino, na katero `ip` kaže, prav tako kot

```
++*ip
```

in

```
(*ip)++
```

V zadnjem primeru so oklepaji potrebni. Brez njih bi izraz povečal kazalec namesto vrednosti, na katero kazalec kaže, ker unarni operatorji kot so `*` in `++` asociirajo z desne proti levi.

Končno, ker so kazalci spremenljivke, jih lahko uporabljamo brez dereferenciranja. Če je `iq` še en kazalec na `int`, tedaj prirejanje

```
iq = ip;
```

prepiše kazalec `ip` v `iq` in sedaj kaže kazalec `iq` na to, na kar je kazal kazalec `ip`.

5.1.3 Kazalčni argumenti funkcij

Ker `C` prenaša parametre po vrednosti, ni direktne poti, da bi funkcija spremenila vrednost kakšne spremenljivke v klicajoči funkciji. V funkciji za sortiranje bi morda radi uporabili funkcijo `swap`, da bi izmenjali dve vrednosti:

```
void swap(int x, int y)     /* NAPAK! */
{
    int     temp;

    temp = x;
    x = y;
    y = temp;
}
```

Funkcija `swap` izmenja *kopiji* spremenljivk ne pa originalov, kot smo nameravali. To dosežemo, če funkciji `swap` pošljemo dva kazalca,

```
swap(&a, &b);
```

funkcijo `swap` pa moramo definirati kot

```
void swap(int *px, int *py) /* interchange *px and *py */
{
    int     temp;

    temp = *px;
    *px = *py;
    *py = temp;
}
```

5.1.4 Zlorabljanje kazalčnih argumentov

Kazalčnih argumentov ne smemo zlorabljati. Funkciji lahko podamo kazalčni argument z namenom, da bo funkcija spremenila vrednost, na katero kazalec kaže. To ni vedno pametna rešitev. Funkcija lahko vrne tudi vrednost in to vrnjeno vrednost lahko izkoristimo, da posredujemo izračunano vrednost klicatelju, namesto da bi klicatelj funkciji podal (morda dodaten) kazalčni argument, ki ga bo potem funkcija uporabila, da bo spremenila vrednost, na katero tak kazalec kaže. Seveda, če funkcija že vrne kakšno vrednost, tedaj ne more vrniti še ene vrednosti. Tak primer je funkcija `getint` v razdelku 5.1.5. Na splošno pa kažejo izkušnje, da začetniki prepogosto uporabljajo nepotrebne kazalčne argumente. Primeri funkcij, ki so navedeni v tej knjigi (ali zastavljeni kot naloge), jasno kažejo, kako je treba funkcije zastaviti.

5.1.5 Funkcija getint

Kazalčni argumenti omogočajo funkciji, da dosega in spreminja objekte v klicajoči funkciji. Za primer sestavimo funkcijo `getint`, ki pretvori zaporedje vhodnih znakov v celo število, eno na klic. Funkcija `getint` mora vrniti pretvorjeno število, obenem pa še signal, ki pove, kdaj zmanjka števil na vhodu. Ne glede na to kakšno vrednost ima `EOF`, to je tudi legalna vrednost prebranega števila. Zato bo funkcija `getint` vrnila status, nič, če je vse v redu, `EOF`, če podatkov ni več, sámo prebrano vrednost pa mora vrniti po drugi poti, recimo s pomočjo nekega kazalčnega parametra.

Naslednji odlomek bo napolnil večkratno vrednost `array` s števili, ki jih bo dekodirala funkcija `getint`:

```
int     n, array[SIZE], getint(int *);

for (n = 0; n < SIZE && getint(&array[n]) == 0; ++n)
    ;
```

Vsak klic funkcije `getint` napolni eno komponento večkratne vrednosti `array`. Pri tem je bistvenega pomena, da funkciji `getint` podamo naslov naše komponente, saj `getint` nima nobene druge možnosti vplivati na vrednosti `array`[2].

Naša funkcija `getint` bo vrnila vrednost `EOF`, če naleti na konec datoteke, vrednost nič, če je funkcija uspešno dekodirala število, in pozitivno vrednost, če je na vhodu zaporedje znakov, ki ne tvori števila.

```
#include <ctype.h>

int getch(void);
void ungetch(int);

/* getint: get next integer from input into *pn */
int
getint(int *pn)
{
    int     c, sign;

    while (isspace(c = getch()))  /* skip white space */
        ;

    if (c != EOF && !isdigit(c) && c != '+' && c != '-') {
        ungetch(c);               /* it is not a number */
        return 1;
    }
```

---

[2] Dobesedno vzeto to ni res. Funkcija `getint` lahko vpliva na komponente večkratne vrednosti `array`, a funkciji `getint` bi morali še povedati, katero komponento naj spremeni.

```
    sign = (c == ’-’) ? -1 : +1;
    if (c == ’+’ || c == ’-’)
        c = getch();
    for (*pn = 0; isdigit(c); c = getch())
        *pn = 10 * *pn + (c - ’0’);
    *pn *= sign;
    if (c == EOF)
        return c;
    ungetch(c);
    return 0;
}
```

V funkciji `getint` smo uporabljali funkciji `getch` in `ungetch`, ker moramo prebrati en znak preveč, da opazimo, da ne spada več k številu.

5.2 Kazalci in večkratne vrednosti

Definirajmo desetkratno število

```
int a[10];
```

Predstavimo ga lahko kot kaže slika 5.1.

Oznaka `a[i]` pomeni `i`-to komponento večkratne vrednosti `a`. Če je `pa` definiran kot kazalec na celo število

```
int *pa;
```

tedaj prirejanje

```
pa = &a[0];
```

pove, da `pa` kaže na element `a[0]`, torej dobimo situacijo, kot jo kaže slika 5.2.

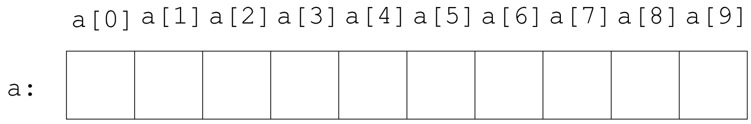
Prirejanje

```
x = *pa;
```

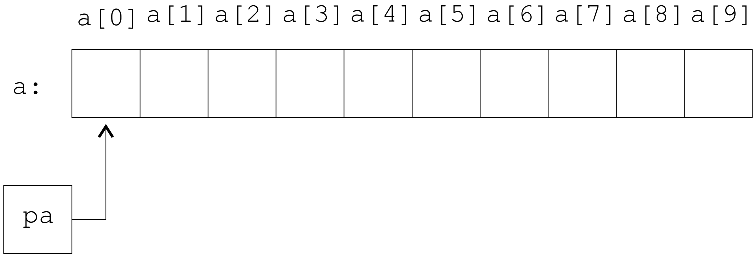
sedaj shrani vrednost `a[0]` v spremenljivko `x`.

Če `pa` kaže na `a[0]`, tedaj po definiciji `pa + i` kaže na element, ki je za `i` oddaljen (naprej) od `a[0]`, to je, kaže na `a[i]`, `pa - i` pa kaže na element, ki je za `i` oddaljen (nazaj) od `a[0]`, to je, kaže na `a[-i]`, če je tak element še vedno del večkratne vrednosti, kar v splošnem primeru ni res.

Ta sklep velja za poljuben tip. Če je `p` nek kazalec, ki nekam kaže, tedaj `p+1` kaže na naslednji element, `p-1` pa na prejšnji element. Brez pomislekov to



Slika 5.1: Večkratna vrednost `int a[10]`



Slika 5.2: Večkratna vrednost `int a[10]` in kazalec `int *pa`

velja le, dokler se sprehajamo po isti večkratni vrednosti. Drugače rečeno, predstavljati si moramo, da si prevajalnik pridrži pravico, da ob definiciji razpostavi objekte po pomnilniku kakor se mu zdi, "z luknjami". Edino za komponente večkratne vrednosti vemo, da slede v pomnilniku neposredno druga drugi.

Ker je ime večkratne vrednosti sinonim za naslov prve (= nulte) komponente, lahko vedno namesto

```
pa = &a[0];
```

pišemo

```
pa = a;
```

Bolj presentljivo je, da lahko namesto

```
a[i]
```

pišemo kar

```
*(a + i)
```

`C` prevajalnik tako in tako interno predela `a[i]` v `*(a + i)` tako, da sta obe obliki res enakovredni.

Če uporabimo adresni operator `&` na obe strani ekvivalence

`a[i]` $\equiv$ `*(a + i)`

dobimo

`&a[i]` $\equiv$ `a + i`

kar je spet res. Pa tudi obratno velja: če je `p` kazalec, je `*(p + i)` isto kot `p[i]`. Seveda pri tem ne smemo pozabiti, da je `p` kazalec, torej spremenljivka, in so izrazi `p = a` in `++p` legalni, `a` pa ni spremenljivka, zato so izrazi kot `a = p` in `++a` ilegalni.

5.2.1 Med kazalcem in večkratno vrednostjo ni razlike?

`C` se zelo trudi, da bi skril razliko med kazalcem in večkratno vrednostjo. Programerju običajno ni treba vedeti ali ima opraviti s kazalcem ali z večkratno vrednostjo, včasih pa je ta razlika življenskega pomena.

Ves problem lahko razložimo na tipu `char`, a vse povedano velja tudi za katerikoli drug tip.

Vemo, da definicija

```
char str[10];
```

definira `str` kot desetkraten znak. Posamezne komponente vrednosti `str` so dostopne kot

```
str[0], str[1], ..., str[9]
```

tako, da je

```
str[i]
```

znak.

Definicija

```
char *p;
```

pove, da je `p` kazalec na znak; ta kazalec še ni inicializiran (razen če ni to statična definicija in v tem primeru so vsi biti tega kazalca nič, kar najbrž pomeni, da je ta kazalec `NULL`). Preden lahko kazalec `p` uporabimo, mu moramo prirediti neko razumno vrednost, recimo

```
p = &str[0];
```

ali, kar je isto,

```
p = str;
```

Od sedaj naprej sta referenci

```
str[4]
```

in

```
p[4]
```

popolnoma legalni in celo pomenita isti znak.

Če sestavimo funkcijo `f` in ji kot parameter podamo `str` ali `p`, funkcija `f` ne more več povedati, ali je kot parameter dobila kazalec na znak ali večkraten znak. Če definiramo

```
void f(char p[])
{
    ...
}
```

ali

```
void f(char *p)
{
    ...
}
```

sta ti dve definiciji popolnoma ekvivalentni. V jedru funkcije se lahko pretvarjamo, da je parameter kazalec ali pa večkratna vrednost, kar nam je v danem trenutku bolj všeč.

5.2.2 Kazalec ni večkratna vrednost

Slika se bistveno spremeni, če uporabljamo globalne spremenljivke. Denimo, da definiramo (globalno) večkratno vrednost

```
char str[10];
```

Taka definicija je globalna, če je napisana zunaj vsake funkcije, vključno zunaj funkcije `main`. Sodeč po deklaraciji je `str` desetkraten znak in ne kazalec na znak kot `p`, če bi uporabili (globalno) definicijo

```
char *p;
```

Dokler ostanemo v datoteki, kjer so naše definicije, je vse, kar smo prej povedali, še vedno res: `str` lahko uporabljamo kot kazalec, `p` pa kot večkratno vrednost. Pa denimo, da imamo še eno izvorno datoteko, kjer deklariramo vrednosti `str` in `p` kot eksterne količine

```
extern char *str;
extern char p[10];
```

Ti dve deklaraciji se očitno ne ujemata z originalnima definicijama in `lint` program nam bo to tudi povedal.

Če sedaj v drugi datoteki, kjer imamo zgornji (napačni) eksterni deklaraciji, napišemo referenco

```
*str
```

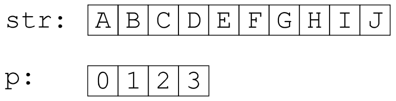
nam bo prevajalnik verjel, da je `str` kazalec, čeprav je v resnici desetkraten znak in bo zgeneriral napačen kod. V prvem primeru je prevajalnik vedel, kako je v resnici s temi rečmi in se ni dal prevariti. V drugem primeru pa bo računalnik pograbil (najbrž) štiri znake iz dejanske desetkratne vrednosti in jih interpretiral kot naslov nekega znaka z grozljivimi posledicami. Ti problemi ne izvirajo iz pisanja `*str` ali `p[4]` ampak iz *nepravilne* deklaracije.

Narišimo si sliko dogajanja. V ta namen predpostavimo, da zasede kazalec štiri zloge, znak pa enega. Dejansko stanje, osnovano na definiciji

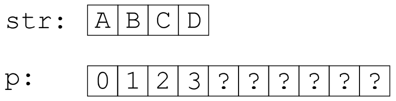
```
char str[10];
char *p;
```

bo videti kot na sliki 5.3, kjer smo skušali prikazati zloge z znaki kot `"ABC..."` in zloge, ki tvorijo kazalec, kot `"0123"`.

V drugi datoteki bo zaradi nepravilnih deklaracij



Slika 5.3: Definicija `char str[10], *p`



Slika 5.4: Deklaracija `extern char *str, p[]`

```
extern char str*;
extern char p[];
```

prevajalnik napravil interpretacijo, ki bo videti kot na sliki 5.4.

V luči povedanega, obnašati se je treba tako, kot Vas je mama učila: "Ne laži!". Če je količina definirana kot večkratna vrednost, jo povsod deklarirajmo kot večkratno vrednost, če je definirana kot kazalec, jo povsod deklarirajmo kot kazalec. Uporabljamo jo pa lahko kot kazalec ali kot večkratno vrednost.

5.2.3 Kazalci kot argumenti funkcij

Če funkciji podamo večkratno vrednost kot argument, bo funkcija videla naslov prvega (nultega) elementa. Znotraj funkcije je tak parameter lokalna spremenljivka, kazalec na nulti znak. To nam omogoča, da napišemo novo verzijo funkcije `mystrlen`.

```
/* mystrlen: return length of string s */
int
mystrlen(const char *s)
{
    int     n;

    for (n = 0; *s; ++s)
        ++n;

    return n;
}
```

Ker je `s` kazalec, je povečanje `s` popolnoma legalno. Izraz `++s` nima nobenega učinka na niz, s katerim smo funkcijo `mystrlen` poklicali. To kar smo povečali, je (privatna) kopija kazalca. To pomeni, da klici kot so

```
size = mystrlen("hello, world");   /* string constant */
size = mystrlen(array);            /* char array[100]; */
size = mystrlen(ptr);              /* char *ptr; */
```

vsi pravilno funkcionirajo.

Kot opis formalnega parametra lahko uporabimo `const char s[]` ali pa `const char *s`. Ljubši nam je kazalec, ker natančneje pove, kaj se dogaja.

5.2.4 Del večkratne vrednosti

Funkciji lahko podamo kot parameter samo del večkratne vrednosti. Če je `a` večkratna vrednost, lahko pošljemo funkciji `f` naslov drugega elementa z

```
f(&a[2])
```

ali

```
f(a + 2)
```

V definiciji funkcije `f` imamo

```
f(int arr[]) { ... }
```

ali pa

```
f(int *arr) { ... }
```

Funkcija `f` nima nobene možnosti, da bi zvedela, da je dobila samo del večkratne vrednosti, ne pa cele, ker dobi pač kazalec na en element.

Če smo gotovi, da taki elementi obstajajo, lahko pišemo tudi `p[-1]`, `p[-2]`, in tako naprej. Element `p[-1]` je tik pred elementom `p[0]`. Razume se, da je nelegalno referencirati elemente izven meja večkratne vrednosti.

Naloga 5.1. Tako, kot je napisana, funkcija `getint` tretira znak `+` ali `-`, ki mu ne sledi števka, kot legalno reprezentacijo števila nič. Popravite `getint`, da bo v takem primeru vrnila znak `+` ali `-` nazaj na vhod.

Naloga 5.2. Sestavite funkcijo `getfloat`, realno varianto funkcije `getint`. Kaj naj `getfloat` vrne kot funkcijsko vrednost?

5.3 Adresna aritmetika

Če `p` kaže na nek element večkratne vrednosti, tedaj z ukazom

```
++p
```

premaknemo kazalec `p` na naslednji element, z ukazom

```
--p
```

na prejsnji element, kazalca

```
p+n
p-n
```

pa kažeta na element n pozicij naprej, oziroma na element n pozicij nazaj.

Če je `q` še en kazalec, ki kaže na nek, v splošnem drug, element kot `p`, še vedno pa oba kazalca kažeta na elemente iste večkratne vrednosti, tedaj je `p-q` število elementov med `*p` in `*q`. To je res za vsak tip. Kazalca `p` in `q` lahko kažeta na elemente večkratne vrednosti, ti elementi pa so poljubnega tipa.

Dejstvo, da lahko kazalce (, ki kažejo v isto večkratno vrednost,) odštevamo, izkoristmo še, da napišemo boljšo verzijo funkcije `mystrlen`:

```
size_t
mystrlen(const char *s)
{
    const char  *p = s;

    while (*p != '\0')
        ++p;

    return p - s;
}
```

5.3.1 Dodeljevanje pomnilnika

Takšne lastnosti kazalčne aritmetike nam omogočajo, da napišemo primitiven dodeljevalnik pomnilnika. Dodeljevalnik bo primitiven v smislu, da bo zahteval, da sproščamo dodeljene kose pomnilnika v obratnem vrstnem redu, kot jih dodeljujemo. Standardna glava `<stdlib.h>` nam ponuja deklaraciji funkcij `malloc` in `free`, ki ne trpita za to boleznijo. Razume se, da vsak malo večji program na veliko uporablja funkciji `malloc` in `free`.

Naši funkciji se bosta imenovali `alloc` in `afree`. Prototipni deklaraciji sta

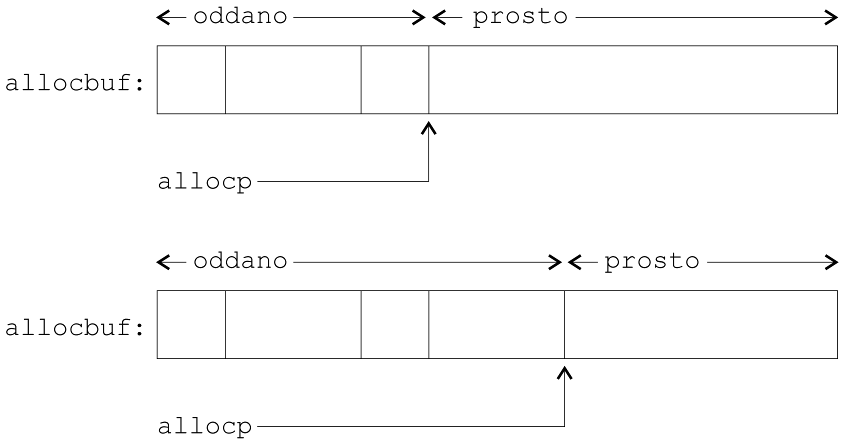
```
void *alloc(size_t size);
void afree(void *ptr);
```

torej `alloc` nekje najde kos pomnilnika velikosti `size` zlogov, in vrne kazalec na začetek dodeljenega bloka, `afree` pa vrne predhodno dodeljen kos pomnilnika nazaj funkciji `alloc`, da ga `alloc` lahko ponovno odda nekomu. Najpreprostejša organizacija je sklad ali LIFO, last-in-first-out konstrukcija.

Na začetku `alloc` od nekod dobi primerno velik kos pomnilnika, recimo `ALLOCSIZE` zlogov. Ta pomnilnik je lahko programu statično dodeljen, lahko ga pa `alloc` zahteva od operacijskega sistema. Naj bo tako ali drugače, `alloc` ve, kje je njegov pomnilnik, in kateri del tega pomnilnika je še prost za dodeljevanje. Na sliki 5.5 je prikazano dogajanje pred četrtim klicem funkcije `alloc` in po njem.

Kot rečeno, `allocbuf` je lahko statično dodeljen kos pomnilnika, lahko je celo brez imena, če prosimo operacijski sistem, da nam podari `ALLOCSIZE` zlogov pomnilnika. Funkcija `alloc` bo poiskala primeren kos (takoj na začetku "prostega" kosa) in vrnila kazalec nanj. Ker `alloc` ne more vedeti, čemu programer potrebuje pomnilnik, `alloc` vrne `void *`, ki ga lahko programer brez prisile prisili v kakršenkoli kazalec.

Funkcija `afree` pograbi navedeni kos pomnilnika in ga vrne v areno, tik pred kazalec `allocp`. Očitno se bo ves program močno sesedel, če `afree` ne bo vračal pomnilnika v sklad v soglasju z disciplino sklada, še bolj pa je napak, če `afree`



Slika 5.5: Dodeljevanje pomnilnika

pokličemo s kazalcem na kos pomnilnika, ki ga nismo dobili od `alloc`. Funkciji `alloc` in `afree` sta napisani v programu A.26 na strani 195.

5.4 Manipulacije z nizi

5.4.1 Konstantni nizi

Konstanten niz, na primer

```
"Jaz sem niz"
```

je niz znakov. V interni reprezenatciji je niz znakov zaključen z znakom `'\0'`, tako da programi lahko najdejo konec niza.

Najpogostejša uporaba konstantnih nizov so parametri funkcij, na primer

```
printf("Hello, world\n");
```

Ko se tak niz pojavi v programu, je dostop do niza realiziran s pomočjo kazalca na začetni znak niza. Zgornji `printf` dobi kot parameter kazalec na začetni znak niza `"Hello, world\n"`.

Konstantni nizi ni treba, da so ravno argumenti funkcij. Če je `pmessage` definiran kot

```
const char *pmessage;
```

potem stavek

```
pmessage = "now is the time";
```

priredi kazalcu `pmessage` večkraten znak. To *ni* prepisovanje niza, samo kazalci so v igri. `C` nima nobenih operatorjev, ki bi procesirali niz znakov kot celoto.

Med definicijama

```
char amessage[] = "now is the time";    /* array of char */
char *pmessage  = "now is the time";    /* pointer */
```

je pomembna razlika. Količina `amessage` je večkraten znak, velik ravno dovolj, da vanj shranimo vse znake v nizu in znak `'\0'`. Posamezne znake te večkratne vrednosti lahko spreminjamo, a cela večkratna vrednost vedno sedi na istem mestu. Količina `pmessage` je kazalec, ki kaže na niz znakov `"now is the time"`. Ta kazalec lahko prestavimo drugam, ne smemo pa spreminjati znakov v nizu `"now is the time"`.

5.4.2 Funkcija mystrcpy

V ilustracijo sestavimo dve koristni funkciji iz standardne knjižnice. Prva funkcija je

```
void mystrcpy(char *s, const char *t)
```

ki prepiše niz, na katerega kaže `t`, na mesto kamor kaže `s`. Funkcija `mystrcpy` je funkcionalno enakovredna funkciji `strcpy`[3] v standardni knjižnici, z dodatkom, da `strcpy` vrne kazalec `s`, ko z delom konča.

Lepo bi bilo, če bi lahko rekli

```
s = t;
```

ampak ta ukaz prepiše kazalec ne pa niza. Če hočemo prepisati niz, potrebujemo zanko. Tu je verzija z večkratnimi znaki.

```
/* mystrcpy: copy t to s; array subscript version */
void
mystrcpy(char s[], const char t[])
{
    int     i;

    for (i = 0; s[i] = t[i]; ++i)
        ;
}
```

Vrednost prirejanja je vrednost leve strani po prirejanju. Zanka se torej ponavlja, dokler prepisuje od `'\0'` različne znake. Ko pa zanka prepiše znak `'\0'`, se zanka konča, s tem pa tudi funkcija.

Tu je verzija s kazalci.

```
/* mystrcpy: copy t to s; pointer version */
void
mystrcpy(char *s, const char *t)
{
    while (*s++ = *t++)
        ;
}
```

---

[3] Pod `UNIX`-om glej `man strcpy`

5.4.3 Funkcija mystrcmp

Druga funkcija, ki si jo bomo ogledali, je

```
int mystrcmp(const char *s, const char *t);
```

Funkcija vrne

`< 0`	če je `s` pred `t`
`> 0`	če je `s` za `t`
`= 0`	če je `s` enak `t`

Vrnjeno vrednost izračunamo kot razliko dveh različnih znakov. Indeksna verzija je

```
/* mystrcmp:
 * return < 0 if s < t, 0 if s == t, > 0 if s > t
 */
int mystrcmp(const char s[], const char t[])
{
    int     i;

    for (i = 0; s[i] == t[i]; ++i)
        if (s[i] == '\0')
            break;
    return s[i] - t[i];
}
```

Kazalčna verzija:

```
/* mystrcmp:
 * return < 0 if s < t, 0 if s == t, > 0 if s > t
 */
int mystrcmp(const char *s, const char *t)
{
    for (; *s == *t; ++s, ++t)
        if (*s == '\0')
            break;
    return *s - *t;
}
```

Glava `<string.h>` vsebuje deklaracije funkcij, ki smo jih obravnavali, in še mnogo drugih[4].

Naloga 5.3. Sestavite kazalčno verzijo funkcije `mystrcat`, ki smo sestavili v poglavju 2.

Naloga 5.4. Funkcija `strend` naj vrne `1`, če se niz `t` nahaja na koncu niza `s`, sicer pa vrne nič.

Naloga 5.5. Sestavite funkcije `mystrncpy`, `mystrncat` in `mystrncmp`, ki delajo kot `mystrcpy`, `mystrcat` in `mystrcmp`, le da upoštevajo samo prvih `n` znakov.

Naloga 5.6. Že sestavljene programe in funkcije iz prejšnjih poglavij prepišite v kazalčne verzije. Kandidati so `getline`, `atoi`, `itoa`, `reverse`, `strindex`, `getop`.

---

[4] Pod UNIX-om glej `man 3c string`

5.5 Večkratni kazalci; kazalci na kazalce

5.5.1 Sortiranje vrstic

Ker so kazalci spremenljivke, jih lahko shranimo v večkratno vrednost tako kot druge objekte. Ilustrirajmo idejo na programu, ki bo posortiral dane vrstice teksta po abecedi, oskubljena verzija Unix programa `sort`[5].

V poglavju 3 smo spoznali funkcijo `shellsort`, ki sortira zaporedje števil po velikosti. V poglavju 4 smo spoznali boljšo metodo, `qsort`, ki prav tako zna presortirati cela stevila. Sedaj moramo `qsort` primerno posplošiti tako, da bo funkcija znala presortirati vrstice, ki so različno dolge in jih ne moremo primerjati in prirejati kot eno samo količino.

Potrebujemo podatkovno strukturo, ki zmore enostavno in učinkovito obravnavati vrstice variabilne dolžine. Posamezne vrstice shranimo v pomnilnik v koščke, ki jih bo dodelil `alloc`. Sedaj potrebujemo samo še večkraten kazalec, ki bo hranil kazalce na začetke nizov, ki tvorijo vrstice. Dve vrstici primerjamo tako, da pošljemo dva kazalca funkciji `strcmp`. Ko bo treba dve vrstici izmenjati, bomo izmenjali raje ustrezna kazalca.

Tako napravimo načrt:

```
read all lines of input
sort them
print them in order
```

Kot običajno je najbolje, če program tako tudi napišemo. Za trenutek odložimo proces sortiranja in se osredotočimo na podatkovne strukture ter vhod ter izhod.

Vhodna funkcija mora zbrati vse znake, ki tvorijo po eno vrstico in zgraditi večkraten kazalec. Prav tako mora vhodna funkcija prešteti število vrstic, saj to stevilo potrebujemo za sortiranje in za izpis. Ker vhodna funkcija prenese samo omejeno število vrstic, lahko ta funkcija vrne število `-1`, če je vhod preobsežen.

Izpisna funkcija samo izpiše vrstice v vrstem redu, določenim s kazalci v večkratni vrednosti. Glej program A.27 na strani 196.

Naloga 5.7. Popravite funkcijo `readlines` tako, da bo `readlines` shranjevala vrstice v velik večkraten znak, ki ga `readlines` dobi od `main`, namesto v pomnilnik, ki ga `readlines` vsakokrat dobi od `alloc`. Koliko hitrejši je popravljen program?

5.6 Večdimenzionalne tabele

`C` pozna večdimenzionalne tabele (matrike), vendar jih skoraj nikoli ne uporabljamo. V redkih primerih, kjer nam večdimenzionalne tabele pridejo prav, so večdimenzionalne tabele konstantnih dimenzij znanih ob prevajanju.

5.6.1 Funkciji day_of_year in month_day

Oglejmo si pretvarjanje datuma. Normalno izražamo datum v obliki `dan`, `mesec`, `leto`, lahko pa tudi v obliki `dan`, `leto`, če `dan` teče od `1` do `365` (ali `366`). Na primer, `1. marec` je `60.`-ti dan v neprestopnem letu, `61.`-ti dan v prestopnem letu. Definirajmo funkcijo `day_of_year`, ki pretvori `dan`, `mesec` in `leto` v dan v letu in funkcijo `month_day`, ki pretvori dan v letu nazaj v `dan` in `mesec`. Ker druga funkcija vrne dve vrednosti, bosta argumenta `m` in `d` kazalca:

---

[5] Pod `UNIX`-om glej `man sort`

```
month_day(1996, 60, &m, &d)
```

postavi `m` na `2` in `d` na `29` (`29. februar`).

Ti dve funkciji potrebujeta isto informacijo, tabelo dni v mesecih ("thirty days hath September ..."). Ker je število dni odvisno od prestopnosti, je najpreprosteje sestaviti dve tabeli, eno za običajno in eno za prestopno leto. Skupaj s testnim programom dobimo izdelek A.28 na strani 199.

Vrednost pogoja, ki jo generira računalnik, je vedno nič (false) ali ena (true). Zato lahko kar pogoj vzamemo za indeks v `daytab`. Elementi so tipa `char` in to je legitimna uporaba tipa `char`.

5.6.2 Večdimenzionalna tabela daytab

Tabela `daytab` je prva večdimenzionalna tabela, ki smo jo srečali. Taka tabela je še vedno večkratna vrednost, le da so njene komponente spet večkratne vrednosti. Elemente referenciramo takole:

```
daytab[i][j]    /* [vrstica][stolpec] */
```

ne pa kot

```
daytab[i,j]     /* NAPAK! */
```

Elementi so shranjeni po vrsticah, tako da najbolj desni indeks variira najhitreje, če elemente dosegamo v takem zaporedju kot so zloženi v pomnilniku.

Večdimenzionalno tabelo inicializiramo temu ustrezno: navedemo seznam komponent v zavitih oklepajih. Seveda je vsaka komponenta spet večkratna vrednost, zato je cela slika rekurzivna. Vsaka vrstica v tabeli `daytab` se začenja z ničlo, da je računanje enostavnejše: številke mesecev lahko tečejo od `1` do `12` namesto od nič do `11`. S tem sicer izgubimo nekaj pomnilnika, pa o tem niti ni vredno razpravljati.

Če pokličemo neko funkcijo in ji podamo naš `daytab` kot argument, tedaj moramo v deklaraciji določiti vse velikosti razen prve, ki je prevajalnik ne potrebuje, na primer

```
void f(char daytab[2][13]) { ... }
```

lahko napišemo kot

```
void f(char daytab[][13]) { ... }
```

podatek `13` pa je potreben.

Navedeno so razlogi, zakaj večdimenzionale tabele v `C` niso popularne. Kasneje bomo videli, kako lahko skonstruiramo popolnoma splošne tabele s pomočjo kazalcev in tedaj bomo videli, kako organiziramo računanje z večdimenzionalnimi tabelami v jeziku `C`.

5.6.3 Funkcija month_name

Zamislimo si nalogo setaviti funkcijo

```
char *month_name(int n)
```

ki vrne kazalec na niz znakov z imenom meseca `n`. To je idealen primer za uporabo statičnih večkratnih vrednosti. Funkcija `month_name` vsebuje privatno zbirko imen mesecev in samo vrne kazalec na pravo ime.

```
/* month_name: return name of n-th month */
char *
month_name(int n)
{
    static char     *name[] = {
        "Illegal month",
        "January", "February", "March",
        "April", "May", "June",
        "July", "August", "September",
        "October", "November", December"
    };

    return name[(n < 1 || n > 12) ? 0 : n];
}
```

5.6.4 Dvodimenzionalna tabela proti kazalcu na kazalec

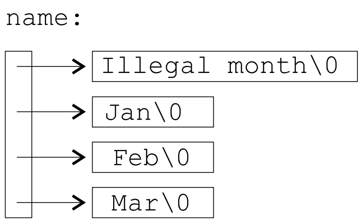
Začetniki v `C` imajo običajno težave z razlikovanjem med dvodimenzinalno tabelo in večkratnim kazalcem. Če definiramo

```
int     a[10][20];
int     *b[10];
```

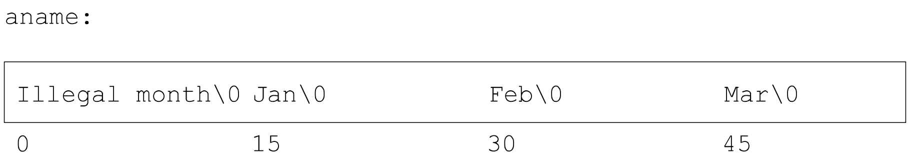
je `a` dvodimenzionalna tabela, za katero je prevajalnik dal na stran prostor za `200 = 10 * 20` celih števil. Običajen algoritem za doseganje elementov je, da je

```
20 * vrsta + kolona
```

recept, kje najdemo element `a[vrsta][kolona]`. V primeru `b` je prevajalnik rezerviral samo prostor za 10 kazalcev, (`int *`), in te kazalce je treba še nekako inicializirati. Če vsak element `b[vrsta]` res pokaže na `20` zaporednih celih števil, tedaj imamo skupaj `10 * 20 = 200` celih stevil in še `10` kazalcev. Zdi se, da je primer `b` bolj razsipen kot primer `a`. Pa temu ni tako. V primeru `b` lahko poskrbimo, da so vrstice različno dolge, vsaka toliko, kot potrebuje, ne pa vse toliko kot najdaljša. Druga, in to ne najmanjša prednost pa je, da funkcijam ni treba nič vedeti o pravokotnih tabelah pač pa le o kazalcih.



Slika 5.6: Definicija `char *name[]`



Slika 5.7: Definicija `char name[][15]`

5.6.5 Tabela imen

Oglejmo si definicijo

```
char *name[] = { "Illegal month", "Jan", "Feb", "Mar", };
```

Kar bi videli v pomnilniku, je prikazano na sliki 5.6.

Če pa definiramo

```
char aname[][15] = { "Illegal month", "Jan", "Feb", "Mar", };
```

bi v pomnilniku videli to, kar kaže slika 5.7.

5.7 Argumenti z ukazne vrstice

V okolju, kjer teče `C`, obstaja način, kako lahko programi vidijo svoje argumente (parametre), potem ko začnejo z delom. Ukaz, s katerim zaženemo program, lupina razdeli na besede, ki so programu dostopne vsaka posebej. Beseda je pri tem poljubno zaporedje znakov ločenih z belimi znaki. Vsak program, vsaka funkcija `main`, dobi dva argumenta, ki se imenujeta `argc` - argument count, in `argv` - argument vector. Argument `argc` je število vseh argumentov. Argument `argv[0]` je ime, s katerim smo program aktivirali, `argv[1]`, `argv[2]`, ... `argv[argc-1]` so imena preostalih nizov, na katere je lupina razbila ukazno vrstico, `argv[argc]` pa je prazen kazalec, `NULL`, tako da lahko sami preštejemo argumente.

5.7.1 Program mygrep1

Za primer si izberimo program `mygrep` iz poglavja 4. Glavna zamera programu je bila, da je niz, ki ga `mygrep` išče, vgrajen v program. `UNIX`-ov program `grep` ima niz, ki ga išče, naveden kot prvi argument in v tej smeri popravimo program `mygrep` v program A.29 na strani 201. Standardna funkcija

```
char *strstr(const char *s, const char *t)
```

vrne kazalec na prvi niz `t` v nizu `s`, `NULL`, če niza `t` v nizu `s` ni. Funkcija `strstr`[6] je deklarirana v glavi `<string.h>`.

5.7.2 Program mygrep2

Program mygrep lahko sedaj posplošimo. Programu hočemo priznati dve neobvezni stikali, `-x` (v pomenu except, razen) in `-n` (v pomenu number, oštevilči). Ukaz `mygrep2 -x -n vzorec` izpiše oštevilčene vse vrstice, ki ne vsebujejo vzorca. Kot je navada v `UNIX`-u, stikala lahko združimo, torej

```
mygrep2 -nx vzorec
```

pomeni isto kot

```
mygrep2 -n -x vzorec
```

Modificiran program A.30 je na strani 202. Količino `argc` zmanjšujemo, količino `argv` pa povečujemo pred vsakim stikalnim argumentom, to je, pred vsakim argumentom, ki se začenja z znakom `'-'`. Na koncu zanke, če ni napak, `argc` pove koliko argumentov ostane neprocesiranih in `argv` kaže na prvega od njih, torej, `argc` mora biti enak ena in `*argv` mora kazati na vzorec. Upam, da ste opazili, da je `*++argv` kazalec na ukazni niz, torej je `(*++argv)[0]` njegov prvi znak. Kot alternativo bi lahko napisali `**++argv`. Ker `[]` veže tesneje kot `*` ali `++`, so oklepaji okrog `*++argv` potrebni. Brez njih bi prevajalnik razumel izraz kot `*++(argv[0])`. To smo v resnici uporabili v notranji zanki, kjer je bil namen sprehoditi se po znakih enega argumenta. V notranji zanki izraz `*++argv[0]` povečuje kazalec `argv[0]`.

Na bolj komplicirane izraze s kazalci zlepa ne naletimo, če pa je kaj takega res potrebno, je pametno razbiti konstrukcijo na nekaj korakov.

5.8 Kazalci na funkcije

Čeprav funkcije niso spremenljivke lahko tvorimo kazalce na funkcije, ki pa so spremenljivke in jih lahko prirejamo, zložimo v večkratno vrednost, vrnemo kot vrednost drugih funkcij in tako naprej. Vse to bomo ilustrirali na primeru sortiranja, ki smo ga napisali prej v tem poglavju. Novost bo, da bo program prepoznal stikalo `-n`, ki terja, da morajo biti vrstice urejene numerično in ne leksikografsko.

Sortiranje je vsekakor sestavljeno iz treh delov:

- primerjanje, ki določi vrsti red para objektov;
- izmenjava, ki izmenja vrstni red dveh objektov;
- sortirni algoritem, ki pa je neodvisen od prvih dveh delov.

---

[6] Pod `UNIX`-om glej `man strstr`

in to se odraža tudi na programu A.31 na strani 204. V klicu `myqsort` sta `strcmp` in `numcmp` dva kazalca na funkcijo. Ker se ve, da sta `strcmp` in `numcmp` funkciji, adresni operator pred funkcijo ni potreben, tako kot ni potreben pred imenom večkratne vrednosti.

Funkcijo `myqsort` smo napisali tako, da lahko dela s kakršnimikoli kazalci, ne samo s kazalci na znake. Kot je razvidno iz prototipa, funkcija `myqsort` pričakuje večkraten kazalec, dve celi števili in funkcijo z dvema kazalčnima argumentoma. Vsi kazalčni argumenti so tipa `void *`. Vsak kazalec lahko prisilimo v `void *` in nazaj ne da bi kaj izgubili. Komplicirana prisila funkcijskega argumenta primerjalne funkcije načelno nima nobenih posledic na dejansko reprezentacijo ampak le dopove prevajalniku, da je vse lepo in prav.

Zadnji parameter funkcije `myqsort` je

```
int (*comp)(void *, void *)
```

ki pravi, da je `comp` kazalec na funkcijo, ki sprejme dva `void *` argumenta in vrne `int`. Uporaba v vrstici

```
if ((*comp)(v[i], v[left]) < 0)
```

je skladna z deklaracijo: `comp` je kazalec na funkcijo, `*comp` je funkcija in

```
(*comp)(v[i], v[left])
```

je klic primerjalne funkcije, ki bo primerjala `v[i]` in `v[left]`. Oklepaji okrog `*comp` so potrebni. Brez njih bi deklaracija rekla

```
int *comp(void *, void*)     /* NAPAK! */
```

to je, da je `comp` funkcija, ki vrne kazalec na `int`, to pa je nekaj drugega.

Če hočemo napisati funkcijo `myqsort` tako, da bo lahko delala s kakršnimikoli elementi, moramo kot dodaten parameter napisati še velikost elementa, da bomo v jedru funkcije `myqsort` lahko računali naslove posameznih elementov.

Naloga 5.8. Funkciji `day_of_year` in `month_day` ne kontrolirata svojih argumentov. Popravite.

Naloga 5.9. Prepišite funkciji `day_of_year` in `month_day` z uporabo kazalcev namesto indeksov.

Naloga 5.10. Sestavite program `expr`, ki izračuna vrednost izraza zapisanega v ukazni vrstici v obratni Poljski notaciji. Na primer, `expr 2 3 4 + *` izračuna `2 * (3 + 4)`.

Naloga 5.11. Modificirajte programa `entab` in `detab`, (ki ste ju napisali v poglavju 1 kot domačo nalogo) tako, da bosta programa sprejela seznam predelčnih pozicij kot argumente. Če pozicije predelčnikov niso predpisane, stopijo v veljavo privzete vrednosti.

Naloga 5.12. Razširite `entab` in `detab` tako, da razumeta okrajšavo

```
entab -m +n
```

ki pomeni predelčnik vsakih `n` kolon začenši v koloni `m`. Izberite (za uporabnika) preprosto privzeto obnašanje.

Naloga 5.13. Sestavite program `mytail`, ki izpiše zadnjih `n` vrstic svojega vhoda. Privzeta vrednost `n` je `10`, lahko jo pa spremenimo s stikalom

```
mytail -25
```

izpiše zadnjih `25` vrstic. Program se mora obnašati razumno ne glede na to kako trapasto vrednost `n` dobi ali ne glede na to kako trapasti so podatki. Vrstice je treba shranjevati kot v sortirnem programu v poglavju 5, ne kot fiksno dvodimenzionalno tabelo znakov.

Naloga 5.14. Modificirajte program `sortlines2` tako, da bo razumel stikalo `-r`, ki pomeni obratni vrsti red. Stikalo `-r` mora funkcionirati tudi skupaj s stikalom `-n`.

Naloga 5.15. Dodajte stikalo `-f` (fold), ki zlije velike in male črke med primerjanjem. Za primerjanje sta potem `a` in `A` ista črka.

Naloga 5.16. Dodajte stikalo `-d` (dictionary order), ki povzroči, da program primerja samo črke, števke in presledke. Stikalo `-d` mora delovati tudi skupaj s stikalom `-f`.

Naloga 5.17. Dodajte koncept polj. Vrstico mora program razumeti kot zaporedje polj, vsako polje je treba ločeno primerjati. Primerjajte `UNIX`-ov program `sort`.

5.9 Dodeljevanje večdimenzionalnih tabel

Pred časom smo dejali, da večdimenzionalne večkratne vrednosti v `C` niso posebno uporabne. Ali to pomeni, da ne moremo delati v matrikami? Seveda ne, le definirati jih ne smemo tako naivno.

Namesto da bi definirali

```
double m[3][4];
```

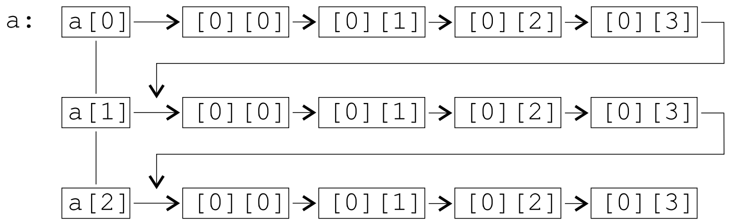
to je, matriko s tremi vrsticami, štirimi stolpci in `double` elementi, s funkcijo `malloc` dodelimo tri kazalce tipa `double *`, od katerih bo vsak kazal na začetek svoje vrstice, te vrstice moramo pa tudi dodeliti. Vse to delo lahko skrijemo v funkcijo

```
double **dmatrix(const long n, const long m);
```

Funkcija `dmatrix` bo dodelila `n` kazalcev tipa `double *`, `n * m` elementov tipa `double` in organizirala kazalce, da bo `i`-ti kazalec kazal na začetek `i`-te vrstice. Za dejansko uporabo potem definiramo

```
double **a;

a = dmatrix(3, 4);
```



Slika 5.8: Rezultat prirejanja `a = dmatrix(3,4)`

in dobimo sliko 5.8.

Matrične elemente lahko sedaj dosegamo normalno: `a[i][j]`.

Sama funkcija `dmatrix` skupaj s formalnim testnim programom je A.32 na strani 207. Dodeljeno matriko lahko vrnemo sistemu s funkcijo `free` tako kot vsako drugo količino, ki smo jo dodelili s funkcijo `malloc`.

Za matrične elemente drugačnega tipa, recimo `float`, potrebujemo drugačne funkcije. Še vedno pa velja pravilo, da se indeksi štejejo od nič naprej.

5.10 Komplicirane deklaracije

Jeziku `C` pogosto očitajo, da so deklaracije po nepotrebnem komplicirane, posebno deklaracije povezane s funkcijami. Deklaracije se skušajo ujemati z uporabo: količino deklariramo (definiramo) tako, kot jo bomo rabili. Pri preprostih deklaracijah ideja funkcionira, pri bolj kompliciranih deklaracijah pa se zaplete, ker deklaracij ne moremo brati z leve proti desni in ker je v njih preveč oklepajev. Razlika med

```
int *f(void); /* f: function returning pointer to an int */
```

in

```
int (*pf)(void);
    /* pf: pointer to function returning an int */
```

ilustrira problem: `*` je predponski operator in ima nižjo prioriteto kot `()`, zato so oklepaji potrebni, da zagotove pravo asociacijo.

Čeprav resnično kompliciranih deklaracij zlepa ne vidimo, je pomembno, da jih znamo razumeti, če že naletimo nanje, in da jih znamo sami sestaviti, če se pokaže potreba. Priporočljiva pot je sestavljanje po kosih z uporabo `typedef` kategorije, kot bomo videli kasneje v razdelku 6.6. Kot pomoč lahko sestavimo preprost program, ki bo prevedel `C` deklaracijo v angleški opis. Angleški opis se seveda lepo bere od leve proti desni. Program `dcl`, glej program A.33 na strani 208, prevede `C` deklaracijo v angleški opis, na primer

```
char **argv
    argv:  pointer to pointer to char
int (*daytab)[13]
    daytab:  pointer to array[13] of int
int *daytab[13]
```

```
        daytab:  array[13] of pointer to int
    void *comp()
        comp:  function returning pointer to void
    void (*comp)()
        comp:  pointer to function returning void
    char (*(*x())[])()
        x:  function returning pointer to array[] of pointer
        to function returning char
    char (*(*x[3])())[5]
        x:  array[3] of pointer to function returning pointer
        to array[5] of char
```

Program `dcl` je zgrajen na osnovi slovnice, ki definira deklarator. V poenostavljeni obliki se slovnica glasi

```
    dcl:        optional *'s direct-dcl
    direct-dcl: name
                (dcl)
                direct-dcl()
                direct-dcl[optional size]
```

ali z besedami, `dcl` je `direct-dcl`, pred katerim ja lahko nekaj zvezdic; neterminalni simbol `direct-dcl` je ime, `dcl` v oklepajih, `direct-dcl`, ki mu sledi par oklepajev ali pa `direct-dcl`, ki mu sledi par oglatih oklepajev z neobvezno velikostjo.

To slovnico lahko uporabimo, da analiziramo deklaracije. Za primer vzemimo deklarator

```
    void (*pfa[])()
```

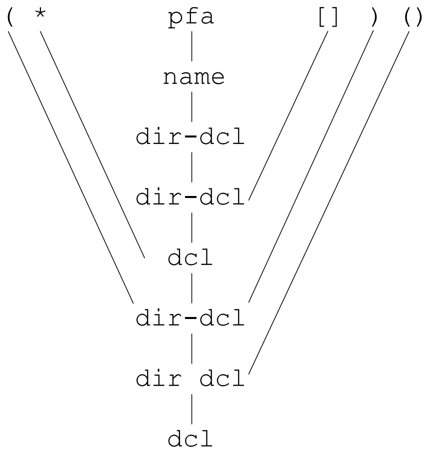
Analizator bo prepoznal `pfa` kot `name`, torej `direct-dcl`, zato bo analizator prepoznal `pfa[]` tudi kot `direct-dcl`. V nadaljevanju bo analizator prepoznal `*pfa[]` kot `dcl`, `(*pfa[])` pa kot `direct-dcl`. Končno bo analizator prepoznal `(*pfa[])()` kot `direct-dcl` in zato tudi kot `dcl`. Ves postopopek je prikazan na sliki 5.9.

V jedru programa `dcl` sta dve funkciji, `dcl` in `dirdcl`, ki analizirata deklaracije v soglasju z navedeno slovnico. Ker je slovnica rekurzivna, se funkciji kličeta rekurzivno, kakor prepoznavata koščke deklaracije. Takemu programu pravimo analizator ali razčlenjevalnik. Program je namenjen za ilustracijo in zato je polno omejitev na deklaracijah, ki jih razume. Program razume samo preproste tipe podatkov kot so `char` in `int`. Funkcije ne morejo imeti argumentov, prav tako program ne razume kvalifikatorjev kot so `const`. Obnašanje v primeru napak je vse prej kot idealno, zato tudi napake v deklaracijah povzroče, da se program zmede. Nekaj teh izboljšav je predlaganih kot naloge.

Funkcija `gettoken` preskoči prazne znake na začetku in potem najde naslednji leksični element na vhodu. Tak leksični element je lahko ime, par okroglih oklepajev, par oglatih oklepajev morda s številom vmes, ali pa en sam znak.

Naloga 5.18. Popravite program `dcl`, da se bo izmotal iz napak v podatkih.

Naloga 5.19. Razširite program `dcl`, da bo razumel deklaracije funkcij s tipi argumentov, kvalifikatorje kot so `const`, in tako naprej.


(*
pfa
[]) ()
name
dir-dcl
dir-dcl
dcl
dir-dcl
dir dcl
dcl

Slika 5.9: Analiza deklaratorja `void (*pfa)[])()`

Poglavje 6

Strukture

6.1 Pojem strukture

Struktura je zbirka ene ali večih spremenljivk, ki so lahko različnih tipov. Te spremenljivke so zbrane v eno enoto, strukturo, da jih je lažje obravnavati kot celoto. V drugih programskih jezikih, kot je Pascal, jim pravijo zapisi.

Tradicionalen primer strukture je plačilni zapis: vsak delavec je opisan z vrsto atributov kot so ime, priimek, naslov, EMŠO, plača, in tako naprej. Vsak od teh atributov, na primer naslov, je spet lahko struktura.

6.1.1 Struktura fraction

Za vzorec si oglejmo strukturo `fraction` za manipuliranje ulomkov. Ulomek tvorita dve celi števili, števec `num` (numerator) in imenovalec `den` (denominator). Ti dve komponenti lahko zložimo v strukturo, ki jo deklariramo takole

```
struct fraction {
    int     num;
    int     den;
};
```

S tem je definiran nov tip, struktura `fraction`, ki ima dve komponenti, `num` in `den`, ki sta tipa `int`. Ključna beseda `struct` začenja deklaracijo strukture, ki je seznam deklaracij v zavitih oklepajih. Ključni besedi `struct` lahko sledi neobvezna značka (tag), `fraction` v našem primeru. Struktura je lahko deklarirana brez značke, a tak primer ni zelo koristen, ker se ne moremo nikoli več nanj sklicevati. Značke uporabljajo svoj prostor imen, zato ima lahko značka tako ime kot spremenljivka.

Spremenljivkam v strukturi pravimo komponente. Tudi imena komponent imajo svoj prostor imen (vsaka struktura svojega), zato se komponente lahko imenujejo enako kot spremenljivke ali značke. Iz konteksta je vedno jasno, ali gre za spremenljivko, značko ali komponento strukture. Seveda te svobode ne zlorabljamo in komponentam v različnih strukturah zlepa ne dajemo istih imen.

6.2 Strukturne spremenljivke

Deklaracija strukture definira nov tip. Zaključnemu zavitemu zaklepaju lahko sledi seznam imen, ki s tem postanejo spremenljivke tega novega tipa. Definicija

```
struct { ... } x, y, z;
```

je sintaktično ekvivalentna definiciji

```
int x, y, z;
```

Če deklaraciji strukture ne sledi seznam spremenljivk, torej

```
struct { ... };
```

tedaj takšna deklaracija definira samo vzorec (obliko) strukture. Če k deklaraciji spada značka, tedaj lahko kdaj kasneje definiramo na primer

```
struct fraction p, q;
```

in s tem definiramo ulomka `p` in `q`.

6.2.1 Inicializacija struktur

Definiciji spremenljivk lahko sledi še seznam začetnih vrednosti v zavitih oklepajih, recimo

```
struct fraction p = { 2, 3 };
```

definira ulomek 2/3. Začetne vrednosti morajo biti konstantni izrazi, celo če je struktura avtomatična.

6.2.2 Doseganje komponent

Komponente strukture dosegamo s konstrukcijo

```
ime-strukture . ime-komponente
```

to je, s piko ločimo ime strukture od imena komponente. Če hočemo izpisati vrednost ulomka `p`, lahko rečemo

```
printf("%d/%d", p.num, p.den);
```

Podobno lahko ulomek `p` pretvorimo v realno stevilo

```
double  pd;

pd = (double)p.num / (double)p.den;
```

(Ena od obeh prisil je odveč).

6.2.3 Gnezdenje struktur

Strukture lahko gnezdimo, to je, komponente strukture so spet lahko (druge) strukture, na primer

```
struct interval {
    struct fraction from;
    struct fraction to;
};
```

Če sedaj definiramo

```
struct interval inter;
```

potem se

```
inter.from.num
```

nanaša na števec `num` ulomka `from` v intervalu `inter`.

6.2.4 Legalne operacije s strukturami

Edine legalne operacije s strukturami so prepisovanje strukture kot celote (prirejanje), uporaba adresnega operatorja `&` in doseganje strukturnih komponent. Prepisovanje (prirejanje) struktur vključuje prenašanje struktur kot parametrov in vračanje strukturnih vrednosti.

6.3 Prenašanje struktur v funkcije

Obstajajo tri različne poti za prenašanje struktur v funkcije:

- prenašanje posameznih komponent,
- prenašanje celih struktur,
- prenašanje kazalcev na strukture.

Vsak od teh načinov ima svoje dobre in svoje slabe strani.

Sestavimo nekaj funkcij za manipuliranje ulomkov. Prva funkcija, ki jo bomo sestavili, bo `makefraction`, ki dobi kot parametra dve celi števili in vrne ulomek, ki ima ti dve števili za števec in imenovalec:

```
/* makefraction:
 * make a fraction from num and den components
 */
struct fraction
makefraction(const int num, const int den)
{
    struct fraction temp;

    temp.num = num;
    temp.den = den;
    return temp;
}
```

Uporaba istih imen za parametre in za komponente je popolnoma legalna.

Funkcijo `makefraction` lahko sedaj uporabimo, da dinamično inicializiramo katerikoli ulomek, ali pa, da skonstruiramo strukturo `fraction` kot parameter funkciji, ki tako strukturo pričakuje.

```
struct interval inter;
struct fraction middle;
struct fraction makefraction(const int, const int);

inter.from = makefraction(0, 1);
inter.to   = makefraction(2, 3);
middle = makefraction(inter.from.num + inter.to.num,
                      inter.from.den + inter.to.den);
```

6.3.1 Aritmetika z ulomki

Naslednji korak so funkcije, ki opravljajo aritmetiko na ulomkih. Običajno zahtevamo od ulomkov, da so okrajšani do kraja in da je imenovalec pozitiven. Temu bomo rekli kanonična oblika ulomka. Podrobnosti so razvidne iz programa A.34 na strani 211.

6.3.2 Kazalci na strukture

Če je treba funkciji podati kot parameter kakšno obsežno strukturo, je navadno pametneje funkciji poslati samo kazalec na strukturo namesto strukture same. Kazalec na strukturo je videti kot kazalec na kakšno drugo spremenljivko. Definicija

```
struct fraction *pp;
```

pravi, da je `pp` kazalec na strukturo `fraction`. V tem primeru je

```
*pp
```

struktura, na katero `pp` kaže,

(*pp).num in (*pp).den

sta komponenti te strukture. Tako lahko pišemo

```
*pp = makefraction(2, 3);
printf("%d/%d\n", (*pp).num, (*pp).den);
```

Oklepaja okrog `*pp` sta potrebna, ker ima operator `.` višjo prioriteto kot operator `*`. Pisano brez oklepajev,

```
*pp.num
```

bi pomenilo

```
*(pp.num)
```

to pa ni prav, ker `pp` ni struktura in iz njega ne moremo izbrati komponente `num`.

Kazalce na strukture uporabljamo tako pogosto, da imamo na voljo krajšo pisavo. Če je `p` kazalec na kakšno strukturo, je

```
p->komponenta
```

okrajšava za

```
(*p).komponenta
```

(operator `->` pišemo kot znak minus `-`, ki mu neposredno sledi znak večji `>`). Pišemo lahko torej

```
*pp = makefraction(2, 3);
printf("%d/%d\n", pp->num, pp->den);
```

Oba operatorja, `.` in `->` asociirata z leve na desno, tako da so v kontekstu

```
struct interval inter, *ip = &inter;
```

izrazi

```
inter.from.num  (inter.from).num
ip->from.num    (ip->from).num
```

vsi ekvivalentni.

6.3.3 Gneča na vrhu prioritetne lestvice

Strukturni operatorji `.`, `->`, klic funkcije `()` in `[]` za indekse, so na vrhu prioritetne lestvice in zato vežejo zelo tesno. Na primer, če imamo definicijo

```
struct {
    int     len;
    char    *str;
} *p;
```

tedaj

```
++p->len
```

poveča `len`, ne `p`, ker so implicirani oklepaji postavljeni takole

```
++(p->len)
```

Z eksplicitnimi oklepaji lahko vezavo spremenimo:

```
(++p)->len
```

poveča `p` preden referencira `len`, in

```
(p++)->len
```

poveča `p` potem, ko je `len` referenciran. (V zadnjem primeru oklepaji sploh niso potrebni).

Podobno,

```
*p->str
```

pograbi to, na kar `str` kaže;

```
*p->str++
```

poveča `str`, potem ko pograbi to, na kar kaže (tako kot `*s++`);

```
(*p->str)++
```

poveča tisto, na kar `str` kaže; in

```
*p++->str
```

poveča `p`, potem ko je pograbil tisto, na kar `str` kaže.

6.4 Večkratne strukture

Zamislimo si program, ki bo preštel, kolikokrat je kakšna ključna beseda referencirana v nekem `C` programu. Potrebujemo večkratnen niz, kjer bodo shranjene ključne besede, in večkratni `int`, kjer bomo šteli, kolikokrat se ključna beseda pojavi. Ena možnost sta dve vzporedni večkratni vrednosti

```
char    *keyword[NKEYS];
int     keycount[NKEYS];
```

Ampak samo dejstvo, da sta večkratni vrednosti vzporedni, sugerira uporabo večkratne strukture. Vsak element je en par

```
struct key {
    char    *word;
    int     count;
};
```

vsega skupaj pa imamo večkraten par. Tako definiramo

```
struct key {
    char    *word;
    int     count;
} keytab[NKEY];
```

to je, strukturo z značko `key`, in večkratno strukturo `keytab`. Vsak element te večkratne vrednosti je struktura. Isto konstrukcijo bi lahko napisali kot

```
struct key {
    char    *word;
    int     count;
};

struct key keytab[NKEY];
```

Ker vsebuje `keytab` konstanten nabor imen, je najbolje, da napravimo `keytab` eksterno spremenljivko, ki jo inicializiramo enkrat za vselej ob definiciji. Inicializacija strukture je analogna prejšnjim primerom: definiciji sledi seznam začetnih vrednosti v zavitih oklepajih:

```
struct key {
    char    *word;
    int     count;
} keytab[] = {
    { "auto",       0 },
    { "break",      0 },
    { "case",       0 },
    { "char",       0 },
    { "const",      0 },
    { "continue",   0 },
    { "default",    0 },
    /* ... */
    { "unsigned",   0 },
    { "void",       0 },
    { "volatile",   0 },
    { "while",      0 },
};
```

Notranjih zavitih oklepajev ni treba pisati, če vsakokrat navedemo vse (obe) komponente, torej:

```
... = {
    "auto",     0,
    "break",    0,
    "case",     0,
    ...
    };
```

vendar je varneje pisati tudi nepotrebne oklepaje.

Število komponent v večkratni vrednosti `keytab` lahko opustimo in prepustimo prevajalniku, da jih prešteje in tako dobimo program A.35 na strani 214.

Naloga 6.1. Navedena verzija funkcije `getword` ne obravnava pravilno podčrtajev, nizov, komentarjev in predprocesorjevih kontrolnih vrstic. Sestavite boljšo verzijo.

6.5 Kazalci na strukture

Da bi si bolje ogledali uporabo kazalcev na strukture, bomo ponovno sestavili program `keyword`, to pot s kazalci. Definicija `keytab` ostane taka kot je bila, `main` in `binsearch` pa je treba spremeniti. Izdelek je program A.36 na strani 217.

V novi verziji je nekaj pomembnih sprememb. Deklaracija `binsearch` mora povedati, da vrne kazalec na `struct key` namesto `int`, in to v deklaraciji prototipa in v dejanski definiciji.

Elemente **`keytab`** dosegamo sedaj s kazalci, kar terja številne spremembe. Količini `low` in **`high`** sta sedaj kazalca na začetek in onkraj konca večkratne vrednosti. Izračun kazalca na sredino ni več enostavno

```
mid = (low + high) / 2; /* NAPAK */
```

ker kazalcev ne moremo seštevati. Ker kazalce lahko odštevamo, dobimo kazalec na sredino z ukazom

```
mid = low + (high - low) / 2;
```

Aritmetika s kazalci upošteva velikosti komponent. Pri tem je treba paziti, da velikost (**`sizeof`**) strukture ni nujno vsota velikosti njenih komponent, ker zaradi zahteve po pravilnem poravnavanju v strukturi lahko nastanejo neimenovane luknje. Če je **`char`** velik en zlog, **`int`** pa štiri, tedaj utegne biti struktura

```
struct {
    char    c;
    int     i;
};
```

gladko velika šest ali celo osem zlogov namesto pet.

6.5.1 Samoreferencirajoče strukture

Denimo, da hočemo rešiti splošnejši problem prešteti primerke vseh besed v nekem tekstu. Ker seznam besed ni znan vnaprej, ga ne moremo posortirati vnaprej in potem uporabiti binarno iskanje. Linearno iskanje pa tudi ne pride v poštev, ker bi trajalo predolgo. Tretja možnost, da bi novo besedo vrinili na pravo mesto in s tem ohranjali urejenost, spet ne pride v poštev, ker vrivanje traja predolgo.

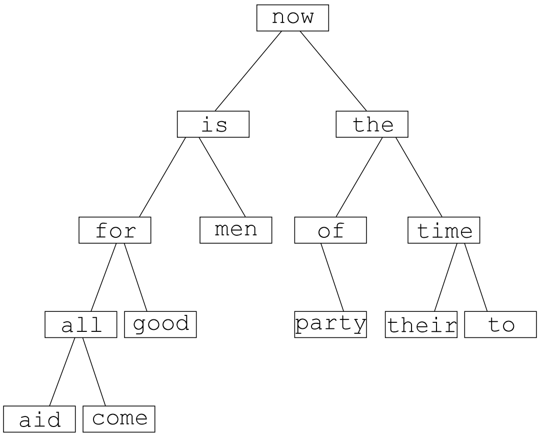
Odgovor je binarno drevo. Binarno drevo vsebuje po en vozel za vsako besedo. Vsak vozel vsebuje

- kazalec na besedo
- število ponavljanj te besede
- kazalec na levo poddrevo
- kazalec na desno poddrevo

Noben vozel ne more imeti več kot dveh potomcev, lahko pa ima samo enega ali pa celo nobenega. Manjkajoče potomce označuje kazalec `NULL`.

Vozle uredimo v binarno drevo tako, da levo poddrevo vsebuje samo besede, ki so leksikografsko pred besedo v vozlu, desno poddrevo pa vsebuje samo besede, ki so leksikografsko po besedi v vozlu. Če vzamemo slavno geslo ameriške republikanske stranke:

```
now is the time for all good men
to come to the aid of their party
```



Slika 6.1: Binarno iskalno drevo

je binarno drevo, ki ga gradimo tako, da vsako novo besedo dodamo v drevo na mesto kamor spada, prikazano na sliki 6.1.

Kako ugotovimo, ali je neka beseda že v drevesu? Začnemo v korenini drevesa. Če se nova beseda ujema z besedo v korenini, tedaj smo besedo že našli. Če se ne ujema, je nova beseda manjša ali večja in iskanje nadaljujemo v levem ali desnem poddrevesu. Če zaidemo v prazno poddrevo, nove besede v drevesu še ni in to prazno poddrevo je ravno mesto, kamor je treba dodati nov vozel. Ta postopek je rekurziven, ker iskanje po poddrevesu uporablja isti kod kot iskanje po drevesu samem. Rekurzivne funkcije za vstavljanje in izpisovanje bodo najbolj naravne.

V skladu z navedenim definiramo

```
struct tnode {                /* the tree node */
    char            *word;  /* points to the text */
    int             count;  /* number of occurrences */
    struct tnode    *left;  /* left child */
    struct tnode    *right; /* right child */
};
```

Na prvi pogled se zdi takšna deklaracija dvomljiva, ker je zagotovo napak, če struktura vsebuje samo sebe. (Dobri vojak Švejk: "V norišnici je bil nekdo, ki je trdil, da je Zemlja votla, notri pa je še ena Zemlja, le da je ta dosti večja od naše.") A naša struktura `tnode` ne vsebuje sama sebe pač pa kazalec na nek (drug) primerek strukture `tnode`, na tem pa ni nič napačnega.

Varianta na isto temo sta dve strukturi, ki se sklicujeta druga na drugo. Takole lahko takšno reč napišemo:

```
struct t {
    ...
    struct s *p;        /* p points to an s */
};
```

```
struct s {
    ...
    struct t *q;        /* q points to a t */
};
```

Napišimo sedaj program, ki bo preštel, kolikokrat se kakšna beseda pojavi v tekstu na standardnem vhodu. To bo program A.37 na strani 220. Funkcija `addtree` uporablja funkcijo `strdup`[1], ki prepiše niz na varno mesto, ki ga dobi od funkcije `malloc`.

Naloga 6.2. Sestavite program, ki prebere nek `C` program, in izpiše po abecedi vse skupine imen, ki se ujemajo v prvih `6` znakih, od nekje naprej pa so različna. Besede v nizih in komentarjih ne štejejo. Število `6` naj bo parameter, ki ga lahko definiramo ob klicu programa.

Naloga 6.3. Sestavite program, ki izpiše vse besede v nekem dokumentu in za vsako besedo seznam vrstic, kjer je beseda referencirana. Program naj opusti prazne besede kot so "the", "and" in tako dalje.

Naloga 6.4. Sestavite program, ki bo izpisal vse različne besede iz nekega teksta. Besede naj bodo urejene po padajoči pogostnosti. Pred vsako besedo naj bo izpisano, kolikokrat se ponovi.

6.5.2 Iskanje po tabelah

V tem razdelku bomo sestavili nekaj funkcij, s katerimi lahko na primer sestavimo makro procesor. Vzemimo na primer ukaz `#define`. Ko makroprocesor naleti na ukaz

```
#define IN  1
```

mora ime `IN` in nadomestni tekst `1` shraniti v tabelo. Ko pozneje naleti na stavek

```
state = IN;
```

mora makro procesor nadomestiti ime `IN` z nadomestnim tekstom `1`.

Vse delo opravljata dve funkciji. Funkcija `install(s, t)` vnese ime `s` in nadomestni tekst `t` v tabelo. Pri tem sta `s` in `t` čisto običajna niza znakov. Funkcija `lookup(s)` preišče tabelo shranjenih imen in vrne kazalec na mesto, kjer sta `s` in `t` shranjena. Če `lookup(s)` imena `s` ne najde, vrne `NULL`.

V tipičnem programu se zlahka nabere nekaj tisoč makrojev, zato je linearno iskanje predrago. Binarno drevo je prekomplicirano. Sestavimo rajši razpršeno tabelo. Vsako ime zgostimo, to je, predelamo v majhno celo število, kar se da slučajno razmetano. Če je to celo število med nič in sto, smo vsa možna imena razdelili v sto razredov, ki imajo sedaj po nekaj deset imen, teh nekaj deset imen pa si že lahko privoščimo, da linearno prečešemo.

Sestaviti moramo torej večkraten kazalec na začetke verig, te verige pa vsebujejo linearne sezname struktur z imeni in definicijami.

---

[1] Pod `UNIX`-om glej `man strdup`

```
struct nlist {
    struct nlist    *next;      /* next entry in the chain */
    char            *name;      /* defined name */
    char            *defn;      /* replacement text */
};
```

Večkraten kazalec bo sedaj

```
#define HASHSIZE    101

struct nlist    *hashtab[HASHSIZE];     /* pointer table */
```

funkcija za zgoščanje pa

```
/* hash: form hash value for string s */
unsigned
hash(const char *s)
{
    unsigned    hashval;

    for (hashval = 0; *s; ++s)
        hashval = *s + 31 * hashval;

    return hashval % HASHSIZE;
}
```

Tip `usigned` nam zagotovi, da je zgoščena vrednost nenegativna. Funkcija `hash` morda ni ravno najboljša, je pa zagotovo kratka in učinkovita. Velikost tabele `HASHSIZE` je, za našo funkcijo `hash`, priporočljiva, da je praštevilo, recimo `101`.

```
/* lookup: look for s in hashtab */
struct nlist *
lookup(const char *s)
{
    struct nlist    *np;

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return np;
    return NULL;
}
```

Zanka v funkciji `lookup` je standarden idiom za sprehod skozi linearno povezan seznam:

```
for (ptr = head; ptr != NULL; ptr = ptr->next)
    ...
```

Funkcija `install` uporablja `lookup`, da ugotovi, če je ime, ki ga mora instalirati, že znano. Če je temu tako, mora nova definicija nadomestiti staro, sicer pa `install` kreira nov element.

```
/* install: put (name, defn) in hashtab */
struct nlist *
install(const char *name, const char *defn)
{
    struct nlist    *np;
    unsigned        hashval;
    struct nlist    *lookup(const char *);

    if ((np = lookup(name)) == NULL) {  /* not found */
        np = malloc(sizeof(*np));
        if (np == NULL || (np->name = strdup(name)) == NULL)
            return NULL;
        hashval = hash(name);
        np->next = hashtab[hashval];
        hashtab[hashval] = np;
    }
    else                                /* already there */
        free(np->defn);
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
}
```

Vse koščke združimo v program macro.c. Program vključuje testni glavni program, ki definira vrednosti makrojev `IN` in `OUT` ter nato izpiše njuni vrednosti. Glej program A.38 na strani 223.

Naloga 6.5. Sestavite funkcijo `undef`, ki odstrani ime in ustrezno definicijo iz tabele, ki jo vzdržujeta `lookup` in `install`.

Naloga 6.6. Sestavite preprost `#define` procesor (brez parametrov), ki uporablja že napisani `install` in `lookup` funkciji.

6.6 Kategorija typedef

`C` ima še en mehanizem, ki se imenuje `typedef`. Deklaracija

```
typedef int Length;
```

deklarira, da je tip `Length` sinonim za tip `int`. Tip `Length` je res ekvivalenten tipu `int`. Uporabljamo ga lahko v deklaracijah, prisilah in tako naprej. To pomeni, da sedaj lahko definiramo

```
Length  len, maxlen;
Length  *lengths[20];
```

Podobno, deklaracija

```
typedef char *String;
```

pravi, da je `String` sinonim za `char *`, torej lahko pišemo

```
String  p, lineptr[MAXLINES], alloc(int);
int mystrcmp(const String, const String);
```

Tip, ki ga `typedef` deklaracija deklarira, stoji na mestu, kjer bi stalo ime spremenljivke. Sintaktično je `typedef` pomnilniški razred (kategorija), tako kot `static` ali `extern`. Deklaracija `typedef` ne napravi novega tipa, napravi samo novo ime za obstoječ tip. V nekem smislu je videti kot makro, vendar ga obravnava prevajalnik in ne predprocesor, zato je močnejši: s `typedef` se da napraviti reči, ki se jih z makroprocesorjem ne da. Po navadi `typedef` imena pišemo z veliko začetnico, da so bolj vidna.

Deklaracije `typedef` nam pomagajo pri kompliciranih deklaracijah, saj nam omogočajo, da komplicirano deklaracijo gradimo s `typedef` po delih. Primer:

```
typedef struct tnode {
    char            *word;      /* points to text */
    int             count;      /* number of occurrences */
    struct tnode    left;       /* left child */
    struct tnode    right;      /* right child */
} Treenode;

typedef Treenode    *Treeptr;
```

S tem smo ustvarili imeni dveh tipov: `Treenode` (struktura) in `Treeptr` (kazalec na strukturo). Sedaj lahko deklariramo

```
static Treeptr addtree(Treeptr, char *);
```

namesto

```
static struct tnode *addtree(struct tnode *, char *);
```

Še bolj je `typedef` koristen v primeru

```
typedef int (*PFI)(char *, char *);
```

ki pravi, da je `PFI` kazalec na funkcijo (dveh `char *` argumentov), ki vrne `int`, in ki jo lahko uporabimo v kontekstu

```
PFI strcmp, numcmp;
```

v programu `sortlines2` v poglavju 5.

Poleg estetskih razlogov pogosto uporabljamo `typedef`, da lažje zagotovimo prenosljivost programov. Tip `size_t` je v več glavah definiran kot `typedef` na primeren tip, recimo `unsigned`. Če želimo spremeniti tip `size_t`, je treba spremeniti samo `typedef` v standardni glavi.

6.7 Unije

Unija je spremenljivka, v katero lahko shranimo, ob različnih časih, spremenljivke različnih tipov. V Pascalu se tej ideji pravi "zapis z variantami".

Osnovna zamisel, čemu uporabljati unije, je, da bi prihranili nekaj pomnilnika. Sintaksa deklaracij unij je sposojena od deklaracij struktur.

Tipična struktura je na primer

```
struct s_tag {
    int     ival;
    float   fval;
    char    *sval;
} s;
```

kjer imamo pod eno streho, v spremenljivki `s`, shranjene (hkrati) količine `ival`, `fval` in `sval`. Velikost (`sizeof`) spremenljivke `s` je vsota velikosti posameznih komponent plus velikost vseh neimenovanih lukenj, ki so potrebne za poravnavanje.

Analogna definicija je

```
union u_tag {
    int     ival;
    float   fval;
    char    *sval;
} u;
```

Količina `u` je unija, ki lahko hrani, ob različnih časih, količine `ival`, `fval` ali `sval`. Velikost (`sizeof`) je velikost največje komponente. Unija je poravnana tako, da je vsaka njena komponenta prav poravnana.

Posamezne komponente unije dosezamo tako kot bi dosezali komponente strukture, torej

```
union-name.member
```

ali

```
union-pointer->member
```

Programer je dolžan paziti, da uporabi tisto komponento, ki jo je nazadnje shranil. Včasih to pravilo namenoma kršimo, da bi pogledali prevajalniku pod krilo, kakšna je videti na primer količina tipa `float`.

```
union {
    float       fval;
    unsigned    uval;
} fu;

fu.fval = 1.5;
(void)printf("\"%g\" is stored as \"%08x\"\n",
    fu.fval, fu.uval);
```

Takšen program seveda ni prenosljiv, lahko ga pa uporabljamo, da odkrijemo, kako so različne vrednosti shranjene v pomnilniku. Seveda uradno tega ne smemo vedeti, ampak radovednost je zelo človeška lastnost.

Če spremenljivka `utype` vodi evidenco o tem, kakšna vrednost je trenutno shranjena v uniji, tedaj utegnemo videti takle, popolnoma legalen, kod:

```
if (utype == INT)
    (void)printf("%d\n", u.ival);
else if (utype == FLOAT)
```

```
        (void)printf("%f\n", u.fval);
    else if (utype == STRING)
        (void)printf("%s\n", u.sval);
    else
        (void)printf("bad type %d in utype\n", utype);
```

Unije se lahko pojavijo v strukturah in večkratnih vrednostih in obratno. Komponente dosezamo kot pri strukturah v strukturah. Na primer, če definiramo večkratno strukturo

```
    struct symtab {
        char    *name;
        int     flags;
        int     utype;
        union {
            int     ival;
            float   fval;
            char    *sval;
        }       u;
    } symtab[NSYM];
```

dosežemo komponento `ival` kot

```
    symtab[i].u.ival
```

prvi znak niza `sval` pa kot

```
    symtab[i].u.sval[0]
```

ali pa

```
    *symtab[i].u.sval
```

Navedeni primer primer napišimo še enkrat. Na videz je ta pot bolj komplicirana, program pa je bolj čitljiv. Zamisel je, da imenom komponent v strukturi, ki je element unije, damo trapasta imena, nato pa razumna imena definiramo s predprocesorjem:

```
    struct symtab {
        char    *name;
        int     flags;
        int     utype;
        union {
            int     _ival;
            float   _fval;
            char    *_sval;
        }       u;
    } symtab[NSYM];

    #define ival    u._ival
    #define fval    u._fval
    #define sval    u._sval
```

Komponento `ival` lahko sedaj uporabljmo takole:

```
symtab[i].ival
```

prvi znak niza `sval` pa kot

```
symtab[i].sval[0]
```

Z unijami smemo početi točno iste reči kot s strukturami: prirejanje, tvorba naslova in doseganje komponent.

Tudi unije lahko inicializiramo ob definiciji, le da smemo pri tem uporabiti le prvo komponento.

6.8 Bitna polja

Bitna polja so drug mehanizem, formalno namenjen za varčevanje s pomnilnikom. Zamislimo si kos nekakšnega prevajalnika, ki mora skrbeti za tabelo imen. Vsako ime nosi s seboj še kakšno informacijo, na primer, ali je ali ni ključna beseda, ali je ali ni zunanji/statičen simbol, in tako naprej. Najbolj kompaktno tako informacijo shranimo kot množico bitov v eni sami `int` ali `char` spremenljivki.

V ta namen običajno definiramo množico mask, ki ustrezajo posameznim bitom, kot na primer

```
#define KEYWORD     01
#define EXTERNAL    02
#define STATIC      04
```

ali pa

```
enum { KEYWORD = 01, EXTERNAL = 02, STATIC = 04 };
```

Števila morajo biti potence števila dva, da lahko te maske lepo kombiniramo. Doseganje bitov sedaj postane preprosto igranje z biti v stilu poglavja 2.

Nekatere idiome pogosto uporabljamo:

```
flags |= EXTERNAL | STATIC;
```

prižge bita `EXTERNAL` in `STATIC` v `int` spremenljivki `flags`, medtem ko ju

```
flags &= ~(EXTERNAL | STATIC);
```

ugasne, in pogoj

```
if ((flags & (EXTERNAL | STATIC) == 0) ...
```

je izpolnjen, če sta oba bita ugasnjena.

Čeprav se takim idiomom kmalu privadimo, `C` ponuja še drugo možnost. Bitno polje je množica sosednjih bitov v besedi pomnilnika. Kaj je beseda, je stvar računalnika in prevajalnika. Tipično je beseda `16` ali `32` bitov, čeprav tega ne smemo vedeti. Zgornje `#define` ukaze lahko nadomestimo z bitnim poljem. Bitno polje ima sintakso (spet) sposojeno od struktur, z dodatkom, da vsaki komponenti sledi še dvopičje `:` in širina polja, merjena v bitih, na primer

```
struct {
    unsigned    is_keyword : 1;
    unsigned    is_extern  : 1;
    unsigned    is_static  : 1;
} flags;
```

Spremenljivka `flags` vsebuje tri eno-bitna polja, `is_keyword`, `is_extern` in `is_static`. Bitna polja vedno jemljemo tipa `unsigned`. Bitna polja se obnašajo kot majhna števila, lahko sodelujejo pri aritmetiki, le samostojne količine niso, zato ne moremo na njih uporabiti adresnega operatorja `&`. Prejšnje primere sedaj lahko napišemo naravneje kot

```
flags.is_extern = flags.is_static = 1;

flags.is_extern = flags.is_static = 0;

if (flags.is_extern == 0 && flags.is_static == 0)
    ...
```

Bitno polje je lahko brez imena, samo dvopičje in širina, za potrebe poravnavanja. Specialno širino `0` uporabimo, da izsilimo poravnavanje na rob naslednje besede.

Skoraj vse v zvezi z bitnimi polji je odvisno od izvedbe. Izvedba odloči, kako široka je beseda, izvedba odloči, ali se smejo bitna polja prelivati iz besede v besedo. Izvedba odloči, ali se besede polnijo z leve na desno ali z desne na levo. Skratka, zagotovo se ne moremo zanašati, da bo bitno polje, zapisano z eno izvedbo, pomenilo isto na drugi izvedbi.

Bitna polja pogosto uporabljamo, da dosegamo dele besed, ki so definirani z računalniško opremo. Na osnovi povedanega so taki programi inherentno neprenosljivi.

Poglavje 7

Branje in pisanje

7.1 Preprosto branje in pisanje

Branje in pisanje uradno ne spadata v jezik `C`, ki o teh vprašanjih sploh nič ne ve. Res pa je, da programi komunicirajo s svojim okoljem še na mnogo bolj komplicirane načine, kot smo videli v dosedanjih programih. `ANSI C` k sreči predpisuje še standardne knjižnice, ki vsebujejo funkcije za opravljanje vseh naštetih nalog, in še mnogo več, čeprav ni treba, da uporabljamo `UNIX` operacijski sistem.

`ANSI` standard precizno definira, katere funkcije obstajajo v standardnih knjižnicah in kako se te funkcije vedejo. Programi, ki uporabljajo samo standardne knjižnice, so prenosljivi: na drugem računalniku in pod drugim operacijskim sistemom jih je treba samo prevesti.

`ANSI C` je uzakonil preprost model tekstovnih datotek. Tekstovno datoteko tvori zaporedje znakov, ki je z znaki za novo vrsto, `'\n'`, razdeljeno na vrstice. Če operacijski sistem ne deluje tako, bo standardna knjižnica poskrbela, da se bo zdelo, da se obnaša tako. Po potrebi bo na primer standardna knjižnica med branjem predelala zaporedje `CR LF` v znak `'\n'` in med pisanjem znak `'\n'` v zaporedje `CR LF`.

7.1.1 Funkcija getchar

Najpreprostejša funkcija za branje je

```
int getchar(void)
```

ki prebere s standardnega vhoda (običajno tipkovnica) po en znak naenkrat, ali pa `EOF`, če naslednjega znaka ni več. `EOF` je makro, definiran v standardni glavi `<stdio.h>` in ima običajno vrednost `-1`, čeprav tega ne smemo vedeti.

V večini okolij, pod `UNIX`-om na primer, nam lupina omogoča, da preusmerimo standarden vhod na kakšno datoteko. Če program `prog` uporablja `getchar`, lahko program `prog` poženemo z ukazom

```
prog < infile
```

ki bo aktiviral program `prog` tako, da bo standardni vhod zanj prihajal z datoteke `infile`. Program `prog` se pri tem ne zaveda, da je lupina prestavila standardni vhod. Specialno, program `prog` ne vidi znakov `< infile` kot argumentov. Podobno lahko standardni vhod prihaja kot standardni izhod nekega drugega programa z mehanizmom pip. Ukaz

```
otherprog | prog
```

na sistemih, ki podpirajo pipe, požene `otherprog` in njegov standardni izhod pošlje "po pipi" na standardni vhod programa `prog`. Tudi v tem primeru je to dogajanje za `prog` in `otherprog` nevidno. Program `prog` še vedno bere znake s funkcijo `getchar`, kot da preusmerjanja in pip ni.

7.1.2 Funkcija putchar

Funkcijo

```
int putchar(int)
```

uporabljamo za pisanje: `putchar(c)` zapiše znak, ki je shranjen v spremenljivki `c`, na standardni izhod, običajno ekran terminala, lahko pa tudi kam drugam, če v lupini uporabljamo preusmerjanje izhoda

```
prog > outfile
```

ali pa pipe

```
prog | anotherprog
```

Rezultati, ki jih izpišemo s funkcijo `printf`, ravnotako najdejo pot na standardni izhod. Vsaj za razumevnja dogajanja si je treba predstavljati, da `printf` predela argumente v zaporedje znakov, ki jih potem s funkcijo `putchar` izpiše na standardni izhod.

7.1.3 Glava stdio.h

V vsaki izvorni datoteki, kjer uporabljamo standardno knjižnico, moramo vključiti standardno glavo `<stdio.h>` z ukazom

```
#include <stdio.h>
```

Ošiljeni oklepaji < in > pomenijo, da je treba datoteko `stdio.h` poiskati na "standardnem mestu"[1].

7.1.4 Program lower

Za programe, ki berejo eno samo datoteko in pišejo eno samo datoteko, so funkcije `getchar`, `putchar` in `printf` dovolj, vsekakor pa dovolj, da lahko začnemo. Vzemimo na primer program `lower`, ki predela vse velike črke na standardnem vhodu v male črke, ostale znake pa prepiše kot so.

---

[1] Na `UNIX` operacijskem sistemu je to imenik `/usr/include`

```
#include <stdio.h>
#include <ctype.h>

/* lower: convert input to lower case */
int
main(void)
{
    int     c;

    while ((c = getchar()) != EOF)
        (void)putchar(tolower(c));

    return 0;
}
```

Funkcija `tolower`[2] je deklararirana v glavi `<ctype.h>`. Funkcije, deklarirane v `<stdio.h>` in `<ctype.h>` so pogosto izvedene kot makroji, da bi se izognili (majhni) izgubi časa pri klicu funkcije. Ne glede na to, ali so `<ctype.h>` funkcije izvedene kot makroji ali kot funkcije, nas uporaba `<ctype.h>` ščiti pred tem, da bi morali poznati detajle znakovnega koda.

7.2 Funkcija printf

Funkcija `printf` prevaja interne binarne vrednosti v znake, ki jih potem izpiše. Tu je, še vedno nepopoln, opis funkcije `printf`

```
int printf(const char *format, ...)
```

Funkcija `printf` prevede interne binarne vrednosti, identificirane z argumenti, ki slede argumentu `format`, v znake in te znake izpiše na standardni izhod. Vrednost funkcije je število izpisanih znakov.

Formatni niz vsebuje objekte dve tipov: običajne znake, ki jih `printf` enostavno prepiše na standardni izhod in formatna določila, od katerih vsako povzroči pretvarjanje in izpis naslednjega argumenta funkcije `printf`. Vsako formatno določilo se začenja z znakom `%` in konča s formatnim znakom. Med znakom `%` in formatnim znakom so lahko, v tem vrstnem redu

- Modifikatorji (v poljubnem vrstem redu), ki modificirajo določilo:
  - Znak minus, ki predpisuje levo poravnavanje prevedenega argumenta.
  - Znak plus, ki predpisuje, da mora biti število vedno izpisano s predznakom.
  - Presledek, ki pravi, da se mora število začeti s presledkom ali z minusom.
  - Ničla, ki določa za numeričen argument popolnitev z vodilnimi ničlami.
- Število, ki določa minimalno širino polja. Preveden argument bo zasedel vsaj toliko znakov. Če ima preveden argument manj znakov kot predpisuje širina polja, bo popolnjen na levi (na desni, če je bilo zahtevano levo poravnavanje) do širine polja. Znak za popolnitev je običajno presledek, ničla, če smo to zahtevali z modifikatorjem.
- Pika, ki ločuje minimalno širino polja od natančnosti.

---

[2] Pod `UNIX`-om glej `man tolower`

Tabela 7.1: Formatni znaki funkcije `printf`

znak	tip argumenta	vrednost izpisana kot
`d`, `i`	`int`	Desetiško število.
`o`	`unsigned`	Nepredznačeno osmiško število (brez vodilne ničle).
`x`, `X`	`unsigned`	Nepredznačeno šestnajstiško število (brez vodilnega `0x` ali `0X`) z uporabo `abcdef` ali `ABCDEF` za števke `10`, ..., `15`.
`u`	`unsigned`	Nepredznačeno desetiško število.
`c`	`int`	En sam znak.
`s`	`char *`	Znaki iz niza dokler ne dosežemo konca niza ali ni izpisano dovolj znakov (natančnost).
`f`	`double`	`[-]m.dddddd`, kjer je število števk `d` določeno z natančnostjo. Natančnost nič zatre decimalno piko.
`e`, `E`	`double`	`[-]m.dddddd{eE}{+-}xx`, kjer je število števk `d` določeno z natančnostjo. Natančnost nič zatre decimalno piko.
`g`, `G`	`double`	Uporabi `e` ali `E` format, če je eksponent manjši od `-4` ali če je eksponent večji ali enak natančnosti, sicer pa uporabi typef format. Ničle in pike na koncu se ne pišejo.
`%`	—	Izpis znaka `%`.

- Število, natančnost, ki v primeru niza določa maksimalno število znakov, v primeru `e`, `E` ali `f` pretvorbe število števk za decimalno piko, število pomembnih števk za `g` in `G` pretvorbo, in v primeru celega števila minimalno število števk (po potrebi dodaj vodilne ničle).
- Črka `h`, če je število, namenjeno za izpis, tipa `short` ali `unsigned short`, črka `l`, če je število, namenjeno za izpis, tipa `long` ali `unsigned long`, črka `L`, če je število, namenjeno za izpis, tipa `long double`.

Formatni znaki, ki ji `printf` razume, so prikazani v tabeli 7.1.

Širino polja in/ali natančnost lahko predpišemo kot `*`. V tem primeru bo `printf` vzel naslednji argument, ki mora biti tipa `int`, in ga uporabil kot širino polja ali natančnost.

Večino formatnih določil smo videli že prej. Edina posebnost, ki je še ne poznamo, je natančnost pri izpisu nizov. Denimo, da hočemo izpisati niz `hello, world` (12 znakov). V naslednji tabeli imamo formatno določilo in rezultat, vklenjen med dvopičja, presledki pa so označeni z ␣, da je razvidna širina polja.

```
%s      :hello,␣world:
%10s    :hello,␣world:
%.10s   :hello,␣wor:
%-10s   :hello,␣world:
```

```
%.15s     :hello,␣world:
%-15s     :hello,␣world␣␣␣:
%15.10s   :␣␣␣␣␣hello,␣wor:
%-15.10s  :hello,␣wor␣␣␣␣␣:
```

Ob tem je na mestu svarilo. Funkcija `printf` uporablja prvi argument, da se odloči, koliko argumentov sploh je in kakšnega tipa so. Če argumentov ni dovolj ali če so napačnega tipa, dobimo lahko izredno čudne (nepravilne) rezultate. Prav tako se je treba zavedati razlike med klicema:

```
printf(s);
```

ki ne dela prav, če niz `s` vsebuje znak `%`, ki ni namenjen za formatno določilo, in

```
printf("%s", s);
```

ki vedno deluje prav.

Funkcija `fprintf`

```
int fprintf(FILE *fp, const char *format, ...);
```

prevaja enako kot `printf`, le prevedene znake zapisuje na datoteko `fp`, namesto da bi jih izpisala na standardni izhod.

Funkcija `sprintf`

```
int sprintf(char *string, const char *format, ...);
```

prevaja enako kot `printf`, le prevedene znake shrani v niz `string`, namesto da bi jih izpisala na standardni izhod. Niz `string` mora biti dovolj velik, da lahko shrani vse generirane znake.

7.3 Variabilne liste argumentov

Funkcija `printf` je prototip funkcije (ne v formalnem smislu) z variabilnim številom argumentov. V `K&R C` je bil ta koncept preprost. `K&R C` se ne meni, če je funkcija poklicana z odvečnimi argumenti, če pa je poklicana s premalo argumenti, je program pač napačen in se bo v najboljšem primeru zrušil. V takšnih razmerah je razmeroma preprosto sestaviti funkcijo, kot je `printf`, kjer prvi argument, `format`, odloča, koliko in kakšni so še dodatni argumenti.

Z `ANSI C` so se variabilne liste argumentov zamotale. `ANSI C` hoče vedeti ali ima funkcija res variabilno število argumentov ali pa je to napačen klic. `ANSI C` pravi, da morajo funkcije z variabilnim številom argumentov to eksplicitno deklarirati s tropičjem `...`, na primer

```
int printf(const char *format, ...);
```

Ta deklaracija pove, da ima `printf` enega ali več parametrov, drugače rečeno, funkcija ima parameter `format` in morda še kaj dodatnih, neimenovanih, parametrov. Taka funkcija mora imeti vsaj en običajen, imenovan parameter, za vsemi običajnimi parametri pa stoji tropičje, ki označuje dodatne neimenovane parametre. Ti dodatni parametri ne samo da nimajo imen, tudi tipa nimajo predpisanega. V resnici `ANSI C` obravnava take parametre kot `K&R C`: ob klicu se tipi `char` in `short` avtomatično razširijo v `int`, tip `float` pa v `double`.

7.3.1 Funkcija minprintf

Sestavimo funkcijo `minprintf`, ki ima prototipno deklaracijo

```
void minprintf(const char *fmt, ...);
```

Funkcija `minprintf` ne vrne vrednosti, ker se nam ne da šteti, koliko znakov bo izpisala. V resnici `minprintf` ne bo izpisala ničesar ampak bo za izpis uporabila standarden `printf`. Ves čar funkcije `minprintf` je ilustracija, kako se sprehodimo skozi variabilno listo parametrov. V ta namen so v glavi `<stdarg.h>` definirani makroji, s katerimi se sprehajamo skozi take variabilne liste.

Tip `va_list` (variable argument list) uporabljamo za definicijo spremenljivke, ki se bo nanašala na vsak variabilen parameter. V funkciji `minprintf` se bo ta spremenljivka imenovala `ap`, argument pointer. Makro `va_start` inicializira `ap`, da pokaže na prvi neimenovan parameter. Makro `va_start` moramo poklicati natanko enkrat preden začnemo `ap` uporabljati. Funkcija mora imeti vsaj en imenovan parameter in makro `va_start` moramo poklicati z zadnjim imenovanim parametrom, da bo `ap` prav nastavljen.

Makro `va_arg` vrne en argument in poskrbi, da `ap` pokaže na naslednjega. Preden končamo, moramo poklicati še makro `va_end`, ki bo počistil za seboj, karkoli je že potrebno.

Na osnovi tega lahko že sestavimo funkcijo `minprintf`, skupaj s testnim programom. Glej program A.39 na strani 226. Kot rečeno, bo dejansko delo opravljala prava funkcija `printf`.

Ob variabilnih listah argumentov omenimo še funkcije `vprintf`, `vfprintf` in `vsprintf`

```
#include <stdio.h>
#include <stdarg.h>

int vprintf(const char *format, va_list ap);
int vfprintf(FILE *fp, const char *format, va_list ap);
int vsprintf(char *s, const char *format, va_list ap);
```

Kot pri `printf` funkciji, `vprintf` piše na standardni izhod, `vfprintf` na datoteko `fp`, `vsprintf` pa v niz `s`. Informacija o tem, kaj naj funkcija izpiše, pa prihaja iz variabilne liste `ap`. Pogosto so funkcije `printf`, `fprintf` in `sprintf` samo ovitki za funkcije `vprintf`, `vfprintf` in `vsprintf`, to je, da je `printf` funkcija izvedena kot

```
int printf{const char *format, ...)
{
    va_list ap;
    int n;

    va_start(ap, format);
    n = vprintf(format, ap);
    va_end(ap);
    return n;
}
```

Naloga 7.1. Sestavite program, ki predela velike črke v male ali pa obratno, v odvisnosti od imena, s katerim je bil program poklican (`argv[0]`).

Naloga 7.2. Sestavite program, ki bo izpisal poljuben vhod na inteligenten način. Minimalno mora znati izpisati negrafične znake v osmiški ali šestnajstiški obliki, odvisno od lokalnih navad, ter razlomiti predolge vrste.

Naloga 7.3. Razširite funkcijo `minprintf` tako, da bo znala obdelati več formatnih določil, tako kot `printf`.

Naloga 7.4. Funkcijo `minprintf` napišite brez uporabe funkcije `printf`. To pomeni, da mora `minprintf` sama razstaviti rezultate na znake in jih potem izpisati s funkcijo `putchar`.

7.4 Funkcija scanf

Funkcija `scanf` je analogija funkcije `printf`, vendar za obratno nalogo. Funkcija `scanf` prebira znake na standardnem vhodu in iz njih gradi vrednosti različnih tipov, tako kot `printf` prevaja vrednosti različnih tipov v zaporedje znakov, ki jih funkcija `printf` potem izpiše na standardni izhod.

Funkcija

```
int scanf(const char *format, ...)
```

bere znake s standardnega vhoda in jih dekodira tako, kot določa formatni niz `format`. Rezultate prevajanja `scanf` shranjuje v spremenljivke, ki so identificirane s preostalimi argumenti, ki *morajo* biti kazalci.

Funkcija `scanf` konča z delom, ko izčrpa formatni niz, ali pa, ko se vhodni znaki ne ujemajo več s tem, kar formatni niz od njih pričakuje. Funkcija vrne število uspešno dekodiranih in prirejenih argumentov. Če `scanf` naleti na konec datoteke, funkcija vrne vrednost `EOF` (, ki ni enaka nič). Vrnjena vrednost nič pove, da `scanf` ni uspela prirediti niti enega argumenta. Naslednji klic funkcije `scanf` nadaljuje s prevajanjem tam, kjer je prejšnji klic prenehal.

Formatni niz lahko vsebuje

- Bele znake, ki jih `scanf` ignorira.
- Običajen znak (ne `%`), ki se mora ujemati z naslednjim nebelim znakom na standardnem vhodu.
- Formatno določilo, ki vsebuje, v tem vrstem redu:
  - znak `%`,
  - neobvezno število, ki določa maksimalno širino polja,
  - neobvezen znak `h`, `l` ali `L`, ki določa širino tarče
  - in konverzijski znak.

Formatno določilo določa pretvarjanje naslednjega vhodnega polja. Rezultat se shrani v spremenljivko, na katero kaže naslednji argument, ki mora biti kazalec.

Vhodno polje je definirano kot zaporedje nebelih znakov in sega do naslednjega belega znaka ali pa, dokler širina polja ni izčrpana. To pomeni, da bo funkcija `scanf` preskakovala znake za konec vrste, da bo našla svoje podatke.

Tabela 7.2: Formatni znaki funkcije `scanf`

znak	tip argumenta	vhodni niz
d	`int *`	Desetiško celo število.
i	`int *`	Celo število, ki je lahko zapisano osmiško (vodilna ničla) ali šestnajstiško (vodilna 0x ali 0X).
o	`unsigned *`	Osmiško število (z ali brez vodilne ničle).
u	`unsigned *`	Nepredznačeno celo število.
x	`unsigned *`	Šestnajstiško število (z ali brez 0x ali 0X).
c	`char *`	Znaki. Vhodni znaki, ki so na vrsti, (privzeto eden), se shranijo na naznačeno mesto brez končnega praznega znaka. Običajen preskok belih znakov se ne zgodi. Če želite prebrati naslednji nebel znak, uporabite `%1s`.
s	`char *`	Niz znakov (brez narekovajev). Argument (kazalec) mora kazati na večkraten znak, v katerem je dovolj prostora za vse znake in za zaključni prazen znak.
e, f, g	`float *`	Realno število z neobveznim predznakom, neobvezno decimalno piko in neobveznim eksponentom.
%		Dobeseden znak % brez prirejanja.

Formatni znak določa interpretacijo vhodnega polja. Ustrezni parameter mora biti kazalec, kakor terja semantika jezika `C`.

Formatni znaki, ki jih `scanf` razume, so prikazani v tabeli 7.2.

Pred konverzijskimi znaki `d`, `i`, `n`, `o`, `u` ali `x` lahko napišemo še `h`, ki pomeni, da je argument `short *` namesto `int *` ali pa `l`, ki pomeni, da je argument `long *` namesto `int *`. Podobno lahko pred konverzijske znake `e`, `f` ali `g` napišemo `l`, ki pomeni, da je argument `double *` namesto `float *`, ali pa `L`, ki pomeni, da je argument `long double *` namesto `float *`.

Za primer uporabe prepišimo preprosti kalkulator iz poglavja 4, glej program A.40 na strani 228.

Kot drug primer denimo, da hočemo prebirati vrstice z datumi oblike

```
3 Jan 1995
```

Klic funkcije `scanf` bo videti takle:

```
int     day, year;
char    monthname[20];

(void)scanf("%d %s %d", &day, monthname, &year);
```

Pred `monthname` ni adresnega operatorja, ker je `monthname` že kazalec (na prvi znak imena).

Če so v formatnem nizu običajni, nebeli znaki, se morajo dobesedno ujemati z vhodnimi znaki. Če hočemo prebrati datum oblike `mm/dd/yy`, lahko napišemo kar

```
int     day, month, year;

(void)scanf("%d / %d / %d", &month, &day, &year);
```

7.4.1 Funkciji fscanf in sscanf

Poleg funkcije `scanf` imamo v standardni knjižnici še funkcijo

```
int fscanf(FILE *fp, const char *fmt, ...);
```

s katero lahko beremo s poljubne datoteke, in funkcijo

```
int sscanf(const char *s, const char *fmt, ...);
```

s katero lahko beremo iz niza `s`. Ta funkcija je koristna, če beremo podatke, ki so napisani v formatu, ki ni predpisan. Recimo, da hočemo prebirati datume, ki so napisani v obliki `30 May 1995` ali pa `05/30/95`. Tedaj preberemo vrstico s funkcijo `getline` in jo razderemo s funkcijo `sscanf`:

```
while (getline(line, sizeof(line)) > 0) {
    if (sscanf(line, "%d %s %d",
        &day, monthname, &year) == 3)
        (void)printf(valid: %s\n", line);
                                  /* 30 May 1995 form */
    else if (sscanf(line, "%d / %d / %d",
        &month, &day, &year) == 3)
        (void)printf(valid: %s\n", line);
                                     /* mm/dd/yy form */
    else
        (void)printf(invalid: %s\n", line);
                                      /* invalid form */
}
```

Naloga 7.5. Sestavite privatno verzijo funkcije `scanf` analogno `minprintf`. Za dejansko branje uporabite funkcijo `scanf`.

Naloga 7.6. Sestavite privatno verzijo funkcije `scanf` analogno `minprintf` brez uporabe funkcije `scanf`, pač pa z uporabo funkcije `getchar`.

Naloga 7.7. Prepišite postfixni kalkulator iz poglavja 4, da bo uporabljal `scanf` in/ali `sscanf` funkcijo.

7.5 Druge datoteke

V vseh primerih doslej smo brali s standardnega vhoda in pisali na standardni izhod (izjemoma na standardno napako). V redkih primerih, ko nam to ni zadoščalo, smo se zanašali na lupino, da bo, preden bo zagnala program, prestavila standardni vhod ali standardni izhod.

7.5.1 Struktura FILE in funkcija fopen

Branje standardnega vhoda in pisanje na standardni izhod včasih ne zadoščata. Vzemimo program `cat`[3]. Poleg komplikacij s stikali, ki jih bomo ignorirali, mora program `cat` prepisati vse navedene datoteke na standardni izhod, na primer

```
cat x.c y.c
```

Vprašanje, na katerega moramo odgovoriti, je, kako povezati imena datotek, ki si jih je uporabnik izmislil, s stavki za branje in pisanje.

Pravilo je preprosto. Preden datoteko lahko beremo ali pišemo, jo moramo *odpreti* s standardno funkcijo `fopen`[4]. Funkcija `fopen` vzame zunanje ime kot na primer `x.c`, se pogodi z operacijskim sistemom, ali je prav, da to datoteko beremo ali pišemo, in vrne kazalec na strukturo tipa `FILE`, ki je deklarirana v glavi `<stdio.h>`.

Program, ki hoče brati ali pisati tako datoteko, mora samo definirati spremenljivko tipa `FILE *`, recimo

```
FILE    *fp;
```

in ji prirediti kazalec, ki ga vrne funkcija `fopen`, ki je deklararina kot

```
FILE *fopen(const char *name, const char *mode);
```

Podrobnosti onkraj tega pa nas ne zanimajo, razen tega še, da lahko datoteko *zapremo*, to je, pretrgamo vez med `FILE *fp` in datoteko s funkcijo

```
int fclose(FILE *fp);
```

ki pri pisanju še počisti izpis do kraja. Funkcija `fclose` vrne število nič ali pa `EOF`.

Prvi parameter funkcije `fopen` je ime datoteke, ki jo želimo *odpreti.* Ime datoteke je pač interna stvar operacijskega sistema, pod katerim `C` teče. Drugi parameter funkcije `fopen` je `mode`, niz, ki pove, kaj mislimo z datoteko početi. Dopustne kombinacije so

`"r"`, `"rb"`	odpri za branje
`"w"`, `"wb"`	zbriši obstoječo vsebino ali ustvari za pisanje
`"a"`, `"ab"`	dodaj; odpri za pisanje na koncu ali ustvari za pisanje
`"r+"`, `"r+b"`, `"rb+"`	odpri za branje in pisanje
`"w+"`, `"w+b"`, `"wb+"`	zbriši ali ustvari za branje in pisanje
`"a+"`, `"a+b"`, `"ab+"`	dodaj; odpri ali ustvari za branje in pisanje na koncu

Nekateri operacijski sistemi ločujejo med tekstovnimi in binarnimi[5] datotekami. V takem primeru izberemo kombinacijo, ki vsebuje `"b"`, če je datoteka binarna, sicer pa kombinacijo brez `"b"`.

---

[3] Pod `UNIX`-om glej `man cat`

[4] Pod `UNIX`-om glej `man fopen`

[5] `UNIX` operacijski sistem ne loči med tekstovno in birarno datoteko

7.5.2 Funkciji getc in putc

Ko je enkrat datoteka odprta, potrebujemo funkcije za branje in pisanje. Med njimi sta funkciji `getc` in `putc` najpreprostejši. Funkcija `getc`

```
int getc(FILE *fp)
```

prebere z datoteke `fp` naslednji znak in ga vrne. Če naslednjega znaka ni ali se ga ne da prebrati, funkcija `getc` vrne `EOF`. Funkcija `putc` je obratna funkcija:

```
int putc(int c, FILE *fp)
```

in izpiše znak iz spremenljivke `c` na datoteko `fp`. Funkcija vrne izpisan znak ali `EOF`, če znaka iz tega ali onega razloga ni mogla izpisati.

7.5.3 Datoteke stdin, stdout in stderr

Preden `C` program steče, operacijski sistem poskrbi, da dobi program tri odprte datoteke, standardni vhod, standardni izhod in standardno napako. Te tri datoteke imajo imena `stdin`, `stdout` in `stderr`. Praviloma, če lupina ne uredi drugače, je standardni vhod (`stdin`) tipkovnica terminala, standardni izhod (`stdout`) in standardna napaka (`stderr`) pa ekran terminala.

7.5.4 Funkciji getchar in putchar

Funkciji `getchar` in `putchar` sta lahko definirani s pomočjo funkcij `getc` in `putc` kot

```
#define getchar()   getc(stdin)
#define putchar(c)  putc((c), stdout)
```

kar lahko smatramo za definicijo njune semantike.

7.5.5 Funkciji fscanf in fprintf

Za formatirano branje z drugih datotek in pisanje na druge datoteke uporabljamo funkciji `fscanf` in `fprintf`

```
int fscanf(FILE *fp, const char *format, ...)
int fprintf(FILE *fp, const char *format, ...)
```

7.5.6 Program mycat

Sedaj pa že lahko sestavimo program `mycat`, program A.41 na strani 229, poenostavljeno verzijo programa `cat`. Program `mycat` prepiše vse navedene datoteke na standardni izhod. Če ni nobena navedena, `mycat` prepiše standardni vhod. Obravnavanje napak v programu `mycat` ni idealno. Če program `mycat` neke datoteke iz tega ali onega razloga ne more odpreti, zapiše sporočilo o napaki na konec kopije (prejšnje) datoteke. Če izpis programa pošljemo po pipi v nek drug program, bo ta program videl sporočilo o napaki, ki mu ni bilo namenjeno. V ta namen začne vsak `C` program delo z dvema datotekama, privezanima na ekran terminala, standardni izhod in standardna napaka, in ne z eno samo, standardni izhod. Če že kaj preusmerjamo, je to standardni izhod, standardno napako pa praviloma pustimo privezano na ekran terminala, da tako vidimo sporočila o napakah.

Tako lahko sestavimo bolj *profesionalno* verzijo programa `mycat`, program A.42 na strani 230. V novi verziji pišemo vsa sporočila o napakah na datoteko `stderr`. V sporočilu se program predstavi z imenom, ki ga pobere iz `argv[0]`, to pa je vedno ime, s katerim je bil program zagnan. To je potrebno, če je tak program sredi dolge verige pip, ker brez imena grešnika ne bi vedeli, kateri program v verigi je izpisal napako.

Druga posebnost je funkcija `exit`, ki lahko nadomesti `return` iz funkcije `main`. V funkciji `main` sta stavka `return 1` in `exit(1)` enakovredna. Funkcijo `exit` pa lahko uporabimo globoko v zaporedju klicanih funkcij ne da bi bilo treba izvesti vse `return` v posameznih funkcijah. Funkcija `exit` takoj prekine izvajanje programa. Izraz *takoj* sicer ni čisto na mestu, ker `exit` še vedno pomeni, da bo nekdo pozaprl vse odprte datoteke, to pa pomeni, da bo vsa informacija, ki tiči v izravnalnikih, res izpisana na datoteke.

Posebnost je še funkcija `ferror`, ki pove, če se je na datoteki zgodila kakšna napaka. Med delom nismo stalno kontrolirali izpisa in smo zametavali vrnjene vrednosti. Na koncu se pa le spodobi pogledati, če je na datoteki kakšna napaka. In kako se take napake znebimo? Funkcija

```
void clearerr(FILE *fp);
```

zbriše informacijo o akumuliranih napakah. Funkcija `fopen` interno izvede `clearerr` na začetku in če na koncu `ferror(fp)` ni res, na datoteki `fp` ni bilo nobene napake. Ker v primeru napake program tako ali tako končamo, klic `clearerr` ni potreben, da bi zbrisali napako.

Analogna funkcija funkciji `ferror` je funkcija `feof`, ki pove, če na datoteki čaka `EOF` napaka. Funkcija `clearerr` zbriše `ferror` in `feof` napako.

7.5.7 Funkcija ungetc

Standardna knjižnica vsebuje še funkcijo

```
int ungetc(int c, FILE *fp)
```

ki vrne znak `c` v datoteko, kot da ni bil nikoli prebran. Standardna knjižnica garantira, da na vsaki datoteki lahko vrnemo po en znak. Ta znak bo naslednja bralna funkcija prebrala kot prvega.

7.5.8 Funkciji fgets in fputs

Standardna knjižnica vsebuje funkcijo

```
char *fgets(char *line, int maxline, FILE *fp)
```

ki je približno ekvivalent naši funkciji `getline`. Funkcija `fgets` prebere naslednjo vrsto, vključno z `'\n'` znakom, z datoteke `fp` v večkratno vrednost `line`. Funkcija prebere kvečjemu `maxline - 1` znakov in konča prebrano vrstico z `\0`. Funkcija vrne `line`, če je uspešno prebrala vrsto, in `NULL`, če je prišlo do napake. Funkcija `getline` vrne torej bolj koristno vrednost kot `fgets`.

Za obratno smer imamo funkcijo

```
int *fputs(char *line, FILE *fp)
```

ki izpiše niz `line` na datoteko `fp`. Niz `line` ni treba, da vsebuje znak `'\n'`. Funkcija vrne ničlo, v primeru napake pa `EOF`.

Standard predvideva še funkciji `gets` in `puts` za branje in pisanje nizov s standardnega vhoda oziroma na standardni izhod. Teh funkcij ne priporočamo. Najhujši ugovor je, da ima funkcija `gets` samo en parameter, ime večkratne vrednosti, ne pa še velikosti. To omogoča pisanje riskantnih programov. Eden od najbolj znanih načinov za vlom v `UNIX` operacijski sistem je bil program `sendmail`, ki je uporabljal funkcijo `gets`, ki ne kontrolira, koliko znakov preberemo v večkratno vrednost. Drug ugovor velja za `gets` in `puts`. Za razliko od `fgets` in `fputs` funkcija `gets` zbriše končni `'\n'`, funkcija `puts` ga pa doda. To pomeni, da je semantika funkcij `gets` in `puts` zadosti drugačna od semantike funkcij `fgets` in `fputs`, da funkcij `gets` in `puts` enostavno ne uporabljamo.

Z uporabo funkcije `fgets` zlahka sestavimo funkcijo `getline`, za katero smo dejali, da ima bolj praktično uporabno vrednost

```
/* getline: read a line, return length */
int
getline(char *line, int max)
{
    if (fgets(line, max, stdin) == NULL)
        return 0;
    else
        return strlen(line);
}
```

Naloga 7.8. Sestavite program, ki bo primerjal dve datoteki in izpisal prvo vrstico, kjer se razlikujeta.

Naloga 7.9. Popravite program `mygrep` v poglavju 4 tako, da bo iskal niz v vseh naštetih datotekah, če pa nobene ni, pa na standardnem vhodu. Ali je treba izpisati ime datoteke, če program najde vzorec?

Naloga 7.10. Sestavite program, ki bo natiskal zbirko datotek, vsako na novi strani z naslovom in številko strani za vsako datoteko.

7.6 Razne funkcije

Standardna knjižnica vsebuje zelo veliko funkcij. Tu bomo našteli samo nekatere.

7.6.1 Glava string.h

Programi, ki uporabljajo funkcije za manipuliranje nizov, morajo vključiti standardno glavo `<string.h>`, ki vsebuje deklaracije teh funkcij[6].

`char *strcat (char *s1, const char *s2)`
Prepiši niz `s2` na konec niza `s1`.

---

[6] Pod `UNIX`-om glej `man 3c string`

`char *strncat (char *s1, const char *s2, size_t n)`
Isto, vendar kvečjemu `n` znakov.

`int strcmp (const char *s1, const char *s2)`
Vrni `< 0`, `== 0` ali `> 0`, če je `s1 < s2`, `s1 == s2` ali `s1 > s2`.

`int strncmp (const char *s1, const char *s2, size_t n)`
Isto, vendar kvečjemu `n` znakov.

`char *strcpy (char *s1, const char *s2)`
Prepiši niz `s2` v `s1`.

`char *strncpy (char *s1, const char *s2, size_t n)`
Isto, vendar kvečjemu `n` znakov.

`char *strdup (const char *s1)`
Prepiši niz `s1` na varno mesto.

`size_t strlen (const char *s)`
Izračunaj dolžino niza `s`.

`char *strchr (const char *s, int c)`
Vrni kazalec na prvi `c` v nizu `s` ali `NULL`.

`char *strrchr (const char *s, int c)`
Vrni kazalec na zadnji `c` v nizu `s` ali `NULL`.

`char *strpbrk (const char *s1, const char *s2)`
Vrni kazalec na znak v `s1`, kjer se prvič pojavi kak znak iz `s2`.

`size_t strspn (const char *s1, const char *s2)`
Vrni dolžino vodilnega niza v `s1`, ki je tvorjen samo iz znakov v `s2`.

`size_t strcspn (const char *s1, const char *s2)`
Vrni dolžino vodilnega niza v `s1`, ki je tvorjen samo iz znakov, ki niso v `s2`.

`char *strtok (char *s1, const char *s2)`
Funkcija `strtok` razume niz `s1` kot zaporedje nič ali več enot znakov, ki so ločeni z znaki v `s2`. Prvi klic (s predpisanim `s1`) vrne kazalec na prvo enoto in zapiše ničelni znak za zadnjim znakom prve enote. Funkcija si zapomni kje v `s1` je že bila in pri naslednjih klicih (z `s1` enakim `NULL`) vrne kazalec na naslednje enote, dokler ni nobene enote več in tedaj vrne funkcija kazalec `NULL`.

`char *strstr (const char *s1, const char *s2)`
Vrni kazalec na prvi pojav niza `s2` v nizu `s1` ali `NULL`.

7.6.2 Glava stdlib.h

Standardna glava `<stdlib.h>` vsebuje deklaracije vrste funkcij, med drugim funkcijo `system`[7] in družino funkcij `malloc`.

Funkcija

```
int system(const char *string)
```

izvede ukaz, shranjen v nizu `string`. Funkcija vrne končno stanje tega ukaza. Če je `string` kazalec `NULL`, funkcija `system` preveri, če lupina `sh` obstaja in ima dovoljenje za izvajanje (, to je, ali program teče pod `UNIX` operacijskim sistemom).

---

[7] Pod `UNIX`-om glej `man system`

Družina `malloc`[8] dodeljuje pomnilnik. Na voljo imamo funkcije

```
void *malloc(size_t size);
void *calloc(size_t nelem, size_t elsize);
void *realloc(void *ptr, size_t size);
void free(void *ptr);
```

Funkcija `malloc` vrne kazalec na sveže dodeljen kos pomnilnika velikosti `size` zlogov. Ta pomnilnik ni inicializiran, zlogi, ki nam jih `malloc` da na voljo, morda vsebujejo (našo) staro informacijo, ki je bila v pomnilniku kdaj prej, ko smo rekli `free`. Funkcija `calloc` vrne kazalec na večkraten (`nelem`-kraten) element velikosti `elsize` zlogov. Ta pomnilnik je inicializiran na vse bite enake nič. Funkcija `realloc` spremeni velikost že dodeljenega pomnilnika. Če je nova velikost manjša od originalne, bo sistem vrnil odvečne zloge na koncu funkciji `malloc`; če je nova velikost večja od originalne, bo funkcija `realloc` poskušala obstoječ blok napihniti na novo velikost, če so zlogi, ki staremu bloku sledijo, še na voljo, če ne bo pa funkcija `realloc` dodelila nov blok večje velikosti, prepisala originalen blok vanj ter originalen blok sprostila. Funkcija `free` sprosti navedeni blok pomnilnika, ki ga je morala prej dobiti od funkcije `malloc`, `calloc` ali `realloc`.

7.6.3 Glava math.h

Glava `<math.h>` vsebuje deklaracije matematičnih funkcij[9]. Same funkcije so shranjene v knjižnici `libm.a`, ki jo moramo vključiti v `cc` ukaz kot `cc ... -lm`.

Funkcije, ki so na voljo, so

`double exp(double x)`
eksponentna funkcija $\exp x$.

`double log(double x)`
naravni logaritem $\ln x, x > 0$.

`double log10(double x)`
desetiški (Briggsov) logaritem $\log_{10} x, x > 0$.

`double pow(double x, double y)`
x^y. Če je $x = 0$ in $y <= 0$, ali če je $x < 0$ in y ni celo število, dobimo napako domene.

`double sqrt(double x)`
kvadratni koren $\sqrt{x}, x >= 0$.

`double sin(double x)`
sinus kota v radianih $\sin x$.

`double cos(double x)`
kosinus kota v radianih $\cos x$.

`double tan(double x)`
tangens kota v radianih $tan x$.

---

[8] Pod `UNIX`-om glej `man malloc`

[9] Pod `UNIX`-om glej `man ime` ali `man` man3m manime, če `man ime` ne opiše prave funkcije.

`double asin(double x)`
arcsin x z vrednostjo na intervalu $[-\pi/2, \pi/2]$.

`double acos(double x)`
arccos x z vrednostjo na intervalu $[0, \pi]$.

`double atan(double x)`
arctan x z vrednostjo na intervalu $[-\pi/2, \pi/2]$.

`double atan2(double y, double x)`
polarni kot točke (x, y) na intervalu $(-\pi, \pi]$.

`double sinh(x)`
hiperbolični sinus $\sinh x$.

`double cosh(x)`
hiperbolični kosinus $\cosh x$.

`double tanh(x)`
hiperbolični tangens $\tanh x$.

`double ceil(double x)`
najmanjše celo število $>= x$, vrnjeno kot `double`.

`double floor(double x)`
največje celo število $<= x$, vrnjeno kot `double`.

`double fabs(double x)`
absolutna vrednost $|x|$.

Poglavje 8

Sistemske funkcije

Funkcije v standardnih knjižnicah, ki smo jih do sedaj uporabljali, je moral tudi nekdo napisati. Nekatere od njih so načelno preproste, recimo funkcija `isdigit`[1], nekatere pa so zelo komplicirane. Vzemimo za primer pisanje `printf`. Zlahka si lahko predstavljamo, kako `printf` razstavi dana števila v zaporedja znakov in te znake shrani v nekakšen izravnalnik, a informacijo v tem izravnalniku bo treba prej ali slej izpisati na ekran. Kako? Za take reči se zanašamo na operacijski sistem pod katerim delamo.

Drug primer je dodeljevanje pomnilnika. Spet si lahko predstavljamo kako funkcija `malloc` sodeluje s funkcijo `free`, da daje klicateljem koščke pomnilnika, a prej ali slej mora funkcija `malloc` prositi operacijski sistem, da ji dodeli primerno velik kos pomnilnika, ki ga bo funkcija `malloc` potem upravljala.

Naj bo tako ali drugače, standardna knjižnica se zanaša na operacijski sistem, da bo ta opravljal zanjo najbolj umazano delo. V tem poglavju si bomo ogledali nekaj sistemskih funkcij, ki jih standardna knjižnica uporablja, uporabljamo jih pa lahko tudi sami direkno, če primerna standardna funkcija ne obstaja ali pa če mislimo, da znamo sami delo bolje opraviti. Vse, kar bomo povedali v tem poglavju, velja strogo rečeno le za `UNIX` operacijski sistem, ki se pokorava `POSIX` standardu. V drugih operacijskih sistemih so (lahko) ustrezne funkcije drugačne, a za mnogo operacijskih sistemov je `UNIX` služil kot model operacijskega sistema in sistemske funkcije utegnejo biti zelo podobne `UNIX`-ovim.

8.1 Datoteke

Eden od osnovnih pojmov, ki jih morajo programi obvladovati, je pojem datoteke. Pod `UNIX`-om je datoteka vse: resnična datoteka, vhodno-izhodna naprava, pipa in še marsikaj. Zamisel je ponuditi enakomerna pravila za dostop do teh objektov.

Vsaka datoteka ima ime, pod katerim jo operacijski sistem prepozna. Ta imena so tvorjena iz poljubnih znakov z izjemo poševnice `/`, ki služi za ločilo med imenom imenika in imenom datoteke. Velike in male črke so različne, nealfanumeričnim znakom pa je bolje, da se izogibamo, da ne bi prišli navzkriž z lupino. Da ne bi bilo imen datotek preveč, so imena datotek urejena v drevesno hierarhijo imenikov: na vrhu drevesa je imenik brez imena (ali z imenom `/`, če

---

[1] Pod `UNIX`-om glej `man isdigit`

Vam je ljubše); v tem imeniku so običajne datoteke in drugi imeniki, ki spet lahko vsebujejo druge imenike, in tako naprej. Tako na primer v tej drevesni strukturi obstaja ime

```
/usr/include/sys/ws/chan.h
```

kar pomeni, da v imeniku `/` obstaja imenik `usr`, ki vsebuje imenik `include`, ki vsebuje imenik `sys`, ki vsebuje imenik `ws`, ki vsebuje datoteko `chan.h`. Navedeno ime je polno ime datoteke. Da se izognemo dolgočasnemu naštevanju imen imenikov, obstaja pojem *delovni imenik*, ki je pač nek imenik v hierarhiji imenikov. Če se ime ne začenja s poševnico, recimo

```
egon/C/ch08/doc/all.doc
```

tedaj je mišljeno, da pred začetnim imenom stoji polno ime delovnega imenika s poševnico, tako da spet dobimo polno ime datoteke.

8.1.1 Funkciji read in write

Če v nekem programu hočemo prebrati neko datoteko, jo moramo najprej *odpreti*. Lupina pa poskrbi, da je nekaj datotek že odprtih. To so standardni vhod, standardni izhod in standardna napaka. Odprta datoteka je identificirana z majhnim nenegativnim številom, z deskriptorjem. Standardni deskriptorji so `STDIN_FILENO`, `STDOUT_FILENO` in `STDERR_FILENO`.

Če hočemo z datoteke brati ali nanjo pisati, moramo vedeti njen deskriptor. Tedaj lahko beremo s funkcijo `read` in pišemo s funkcijo `write`.

```
#include <unistd.h>

ssize_t read(int fd, void *buf, size_t nbyte);
ssize_t write(int fd, void *buf, size_t nbyte);
```

Funkcija `read` bo z deskriptorja `fd` skušala prebrati `nbyte` zlogov v izravnalnik `buf` (, ki mora imeti vsaj `nbyte` zlogov prostora) in vrnila število uspešno prebranih zlogov. Funkcija vrne vrednost nič, če ni ničesar prebrala, ker je naletela na konec datoteke. Funkcija vrne `-1`, če je med branjem naletela na napako. V tem primeru bo v `errno` zapisala kodo napake[2]. Vrednost funkcije je tipa `ssize_t`, ki je predznačena oblika tipa `size_t`, da lahko vključuje tudi vrednost `-1`.

Funkcija `write` bo na deskriptor `fd` skušala zapisati `nbyte` zlogov iz izravnalnika `buf` (, ki mora vsebovati vsaj `nbyte` zlogov) in vrnila število uspešno zapisanih zlogov. Funkcija vrne `-1`, če je med pisanjem naletela na napako. V tem primeru bo v `errno` zapisala kodo napake[3].

8.1.2 Program mycp

Oglejmo si program, ki uporablja funkciji `read` in `write`, da prepiše standardni vhod na standardni izhod. Za branje uporabljamo izravnalnik velikosti `BUFSIZ`, definirane v `<stdio.h>` kot priporočljiva velikost izravnalnikov. Program se imenuje `mycp` kot naša verzija programa `cp`[4], glej program A.43 na strani 231.

---

[2] Pod `UNIX`-om glej `man 2 read`

[3] Pod `UNIX`-om glej `man 2 write`

[4] Pod `UNIX`-om glej `man cp`

8.1.3 Funkcija mygetchar

Poučno je videti, kako lahko napišemo funkcije iz standardne knjižnice s pomocjo `read` in `write` funkcij. Tu je na primer funkcija `getchar`, ki smo jo imenovali `mygetchar`, da je ne bi zamenjali s pravo funkcijo `getchar`. Funkcija `mygetchar` mora prebrati en sam znak in vrniti prebrani znak prisiljen v `unsigned char`, da ga return zagotovo prav razširi v `int`, ali pa `EOF`. Glej program A.44 na strani 232.

Funkcija `mygetchar` je preveč neučinkovita: da prebere $80,000$ znakov (na 100 MHz Pentium procesorju) traja $3.12s$. Rezultati so precej boljši, če branje organiziramo v izravnalnik in posredujemo znake iz izravnalnika, dokler jih ne zmanjka in šele tedaj ponovno pokličemo funkcijo `read`, da napolni izravnalnik. Glej program A.45 na strani 233. V resnici, istih $80,000$ znakov prebere nova funkcija v $0.04s$, torej 80 krat hitreje.

8.1.4 Funkcija open

Če hočemo v nekem programu uporabljati druge datoteke, jih moramo najprej *odpreti* s sistemsko funkcijo `open`

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int
open(const char *path, int oflag, ... /* mode_t mode */);
```

Funkcija `open` ima običajno dva argumenta, ime `path` in število `oflag`. Ime `path` je ime datoteke, kot smo ga zgoraj opisali, `oflag` pa pove, kako hočemo datoteko odpreti. Za vrednost `oflag` izberemo enega od makrojev

`O_RDONLY`	odpri samo za branje
`O_WRONLY`	odpri samo za pisanje
`O_RDWR`	odpri za branje in pisanje

lahko pa dodamo (inkluziven ali, `|`) še celo vrsto drugih makrojev[5], od katerih so najbolj zanimivi:

`O_APPEND`	odpri datoteko za dodajanje na koncu datoteke
`O_CREAT`	če imenovana datoteka ne obstaja, jo `open` ustvari; v tem primeru moramo uporabiti še tretji argument, `mode`, ki pove kakšna dovoljenja ima novoustvarjena datoteka[6]
`O_TRUNC`	če datoteka obstaja in jo `open` uspešno odpre za pisanje (O_WRONLY ali O_RDWR), zbriši staro vsebino datoteke

Funkcija `open` vrne majhno nenegativno celo število, deskriptor, ali pa `-1` in tedaj je v globalni spreminljivki `errno` shranjeno število, ki pove, kaj je narobe. S funkcijo `perror`[7]

```
#include <stdio.h>
```

---

[5] Pod `UNIX`-om glej `man open`

[7] Pod `UNIX`-om glej `man perror`

```
void perror(const char *s);
```

izpišemo tekst `s`, dvopičje, presledek, napako in novo vrsto na standardno napako. Napako opisuje sporočilo, ki je zgrajeno na osnovi vrednosti `errno`, kjer je shranjena koda zadnje napake.

Operacijski sistem `UNIX` pozna še arhaično funkcijo `creat`, ki pa je postala odvečna, ker isto funkcionalnost dosežemo z

```
open(ime, O_RDWR | O_CREAT | O_TRUNC, mode);
```

8.1.5 Program mycp1

Sestavimo ilustracijo obravnavanih pojmov in sicer program `mycp1`, ki je poenostavljen program `cp`[8]. Poenostavitev je v tem, da program prepiše samo eno datoteko, drugi parameter ne sme biti imenik in dostopna dovoljenja si kratko in malo izmisli (`rw.r.-`). Kasneje bomo videli, kako lahko preberemo kakšna dovoljenja ima originalna datoteka in kako jih preselimo na novo datoteko. Glej program A.46 na strani 234. Kot zanimivost omenimo (našo) funkcijo `error`

```
void error(const char *fmt, ...);
```

ki izpiše sporočilo o napakah in konča program z `exit(1)`. Funkcija `error` je organizirana kot funkcija `printf`, tako da razume enake argumente kot `printf`.

8.1.6 Funkcija close

`UNIX` tipično dovoljuje kakšnih 20 istočasno odprtih datotek, zato moramo biti pripravljeni, da bomo pri velikem številu datotek stare deskriptorje zapirali, da bomo napravili prostor za nove. To je funkcija `close`[9]

```
#include <unistd.h>

int close(int fd);
```

kjer je `fd` deskriptor, ki ga vrnila funkcija `open`. To zapiranje nima nobenih dodatnih učinkov na izravnalnike, ker izravnalnikov ni.

Še odprte deskriptorje bo zaprl sistem, ko bomo s programom končali (`exit`).

8.1.7 Funkcija remove

Funkcija `remove`

```
#include <stdio.h>

int remove(const char *path);
```

---

[8] Pod `UNIX`-om glej `man cp`

[9] Pod `UNIX`-om glej `man close`

odstrani datoteko (ali prazen imenik) z imenom `path` iz sistema datotek. Strogo vzeto, datoteka bo izginila šele, ko jo bo program zaprl, če je ob `remove` odprta. To pomeni, da lahko odpremo datoteko (z `O_CREAT` opcijo), in jo takoj odstranimo (`remove`). Datoteka bo ostala v sistemu, dokler je ne zapremo (, ali je ne bo zaprl sistem namesto nas). To je ugoden trik za začasne (scratch) datoteke: datoteko odpremo (z `O_CREAT`), jo takoj odstranimo, uporabljamo jo pa naprej in dokler je ne zapremo, nas bo počakala. Ni nam treba skrbeti, kako bomo ob koncu programa take datoteke izbrisali, za to je že poskrbljeno.

Naloga 8.1. Prepišite program `mycat` iz poglavja 7.1, da bo uporabljal `read`, `write`, `open` in `close` namesto standardne knjižnice. Primerjajte učinkovitost obeh verzij.

8.1.8 Funkcija lseek

Branje in pisanje je praviloma zaporedno: vsako branje začne brati tam, kjer je prejšnje branje končalo posel. Včasih pa se želimo sprehajati po datoteki sem ter tja: preberi 10 zlogov začenši na zlogu številka 120, nato pa še 5 zlogov začenši na zlogu številka 20.

Takšno sprehajanje nam omogoča funkcija `lseek`

```
#include <sys/types.h>
#include <unistd.h>

off_t lseek(int fd, off_t offset, int whence);
```

Funkcija `lseek` prestavi trenutni položaj (odprte) datoteke na zlog številka `offset`, šteto, odvisno od `whence`, od

`whence` = `SEEK_SET`	šteto od začetka
`whence` = `SEEK_CUR`	šteto od trenutnega položaja
`whence` = `SEEK_END`	šteto od konca datoteke

Funkcija `lseek` vrne v primeru napake `-1`, `errno` pa vsebuje kod napake. V primeru uspeha pa `lseek` vrne nov položaj datoteke.

Klic

```
lseek(fd, 0, SEEK_SET)
```

previje datoteko `fd` na začetek, funkcija

```
ssize_t get(int fd, off_t pos, void *buf, size_t n)
{
    return (lseek(fd, pos, SEEK_SET) >= 0) ?
        read(fd, buf, n) : -1;
}
```

pa prebere `n` zlogov z datoteke `fd` s pozicije `pos`.

8.1.9 Funkcija fseek

Funkcija `fseek` v standardni knjižnici je ekvivalent funkcije `lseek`

```
int fseek(FILE *fp, long offset, int whence);
```

in vrne `-1` v primeru napake, sicer pa 0.

8.2 Funkcija myfopen in podobno

Za ilustracijo vseh teh pojmov sestavimo privatno verzijo funkcij `fopen` in `getc`. V ta namen potrebujemo seveda tudi privatno verzijo datoteke `<stdio.h>`. Vsa imena, ki se utegnejo križati z obstoječimi imeni, bomo nadomestili z imeni oblike `my...`, da teh nesporazumov ne bo, glej datoteko A.47 na strani 236. Osnovna komponenta je struktura `myiobuf`, ki jo uporabimo v `typedef`, da definiramo `MYFILE` kot analogijo prave strukture `FILE`.

Količina `myiob` je deklarirana kot eksteren večkraten `MYFILE`, `MYFILE *` količine `mystdin`, `mystdout` in `mystderr` pa so definirane kot makroji in sicer kot kazalci na prve tri strukture v `myiob`.

Kot naslednje deklariramo kot eksterni funkciji `myfillbuf` in `myflushbuf` in sicer kot ekvivalenta funkcij `fillbuf` in `flushbuf`. Ti dvi funkciji potrebujemo v makrojih `mygetc` in `myputc`.

Glavo končamo z makrojema `mygetchar` in `myputchar` ter s prototipno deklaracijo funkcije `myfopen`.

Kot naslednje zato potrebujemo `myfopen.c`, glej program A.48 na strani 238. Funkcija `myfopen` najprej preveri, če je prvi znak parametra `mode` razumen, nato pa poišče nezasedeno komponento večkratne vrednosti `myiob`, odpre datoteko tako kot to zahteva `mode` in inicializira strukturo `myiob`.

Naslednja datoteka je `myfillbuf.c`, glej program A.49 na strani 239, ki napolni izravnalnik in vrne prvi znak v njem, da je makro `mygetc` enostavnejši. Funkcija seveda najprej poskrbi, da izravnalnik sploh obstaja.

Podobno potrebujemo funkcijo `myflushbuf`, ki jo boste sami sestavili.

Končno potrebujemo še `myiob.c`, ki definira vse `MYFILE` strukture, kjer smo predpostavili, da so `STDIN_FILENO`, `STDOUT_FILENO` in `STDERR_FILENO` kar števila `0`, `1` in `2`. Glej program A.50 na strani 240.

Naloga 8.2. Prepišite `mystdio.h`, `myfopen.c`, `myfillbuf.c` in `myiob.c` datoteke tako, da bodo uporabljale bitna polja namesto mask. Primerjajte velikost koda in hitrost izvajanja.

Naloga 8.3. Sestavite manjkajoče funkcije `myflushbuf`, `myfflush`, `myfclose` kot analogije funkcij `_flushbuf`, `fflush`, `fclose`.

Naloga 8.4. Sestavite funkcijo `myfseek` s prototipom

```
int myfseek(MYFILE *fp, long offset, int whence);
```

ki je ekvivalent funkcije `lseek`[10].

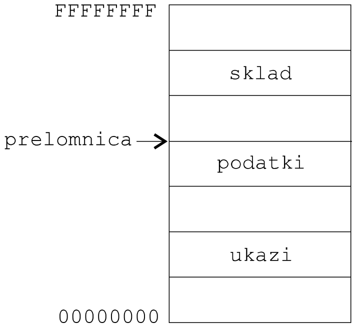
8.3 Dodeljevanje pomnilnika

V poglavju 5 smo napisali primitiven dodeljevalnik pomnilnika in ob tem pripomnili, da bomo raje uporabljali standardni funkciji `malloc` in `free`, ki ne vztrajata na arhitekturi sklada.

V tem poglavju bomo sestavili svojo verzijo funkcij `malloc` in `free`, funkciji `mymalloc` in `myfree`. Ti dve funkciji lahko kličemo v poljubnem vrstnem redu, seveda pri pogoju, da klic `myfree` sledi klicu `mymalloc`, tako kot pri pravi funkciji `malloc` in `free`. To bo realistična uporaba struktur, unij in `typedef` deklaracij.

---

[10] Pod `UNIX`-om glej `man fseek`



Slika 8.1: Razpored informacije v izvedljivem programu

Najprej bomo razrešili vprašanje, od kod naj `mymalloc` dobil pomnilnik, ki ga bo dodeljeval uporabnikom.

Razpored informacije v izvedljivem programu je tak kot ga prikazuje slika 8.1.

Na datoteki z izvedljivim programom je informacija zapisana v segmentih: segment z ukazi programa, podatkovni segment z inicializiranimi spremenljivkami ter BSS segment z neinicializiranimi spremenljivkami. Operacijski sistem zlepi podatkovni segment in BSS segment v en sam podatkovni segment. Naslov tik nad tako zlepljenim podatkovnim segmentom imenujemo prelomnica in ta prelomnica je pomična. Podatkovni segment sega torej od fiksnega začetka do premične prelomnice. Če premaknemo prelomnico navzgor, smo s tem povečali podatkovni segment. Tak premik mora seveda napraviti operacijski sistem, ampak na zahtevo programa. Sistemska funkcija[11]

```
void *sbrk(int incr)
```

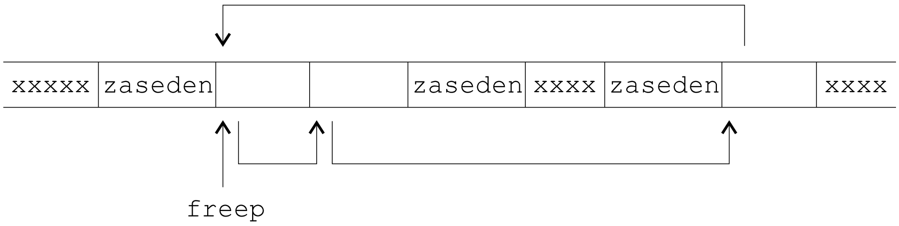
pomakne prelomnico programa navzgor za `incr` in s tem poveča podatkovni segment. Teoretično lahko vzamemo negativen `incr` in s tem pomaknemo prelomnico navzdol, a tega nikoli ne počnemo. Funkcija `sbrk` vrne staro vrednost prelomnice ali pa `-1` in kodo napake v spremenljivki `errno`, če `sbrk` najde kakršnokoli napako.

Argument funkcije `sbrk` je število novih zlogov, vrednost funkcije pa kazalec na novo pridobljen pomnilnik, ki ga mora `mymalloc` sedaj upravljati. To najlažje dosežemo tako, da novi blok pomnilnika prepričamo, da je bil dobljen s kakšnim prejšnjim klicem `mymalloc`, in potem uporabimo `myfree`, da dodamo novi blok v seznam razpoložljivih blokov pomnilnika.

Nikakor ne moremo pričakovati, da bo nov blok pomnilnika, dobljen s klicem `sbrk`, nadaljevanje prejšnjega, ker bodo sistemsko funkcijo `sbrk` medtem klicali tudi drugi deli programa. To sugerira organizacijo pomnilnika, ki ga upravlja `mymalloc`, namreč krožno povezan seznam prostih blokov. Ti prosti bloki so urejeni po naraščajočih naslovih. Shematično so ti bloki prikazani na sliki 8.2. `xxxxx` predstavlja bloke, ki *ne* spadajo pod administracijo funkcije `mymalloc`. Bloki, označeni z `zaseden` sicer spadajo pod administracijo funkcije `mymalloc`, vendar jih je `myalloc` že dodelil nekomu. Bloki označeni s puščicami, so bloki, s katerimi `mymalloc` trenutno razpolaga (prosti seznam).

---

[11] Pod `UNIX`-om glej `man sbrk`



Slika 8.2: Prosti seznam funkcije `mymalloc`

Posamezen blok pod administracijo funkcije `mymalloc` ima na začetku kazalec na naslednji prosti blok v seznamu in število enot (velikost bloka). Enota je definirana kot velikost unije, ki vsebuje poleg pravkar naštetih komponent še spremenljivko najbolj zahtevnega tipa kar se tiče poravnavanja, to je `Align`. V našem primeru smo izbrali, da je `Align long`. Na računalniku z drugačno arhitekturo utegne biti to na primer `double` ali pa `char` (nič zahtev na poravnavanje).

8.3.1 Program xmymalloc

Program `xmymalloc`, A.51 na strani 241, je formalni test program funkcije `mymalloc`, ki v zanki dodeljuje po en megazlog pomnilnika, dokler se `sbrk` ne upre.

Funkcija `mymalloc` uporablja `base` za začetek. Na začetku je `freep` enak `NULL` in to izkoristimo, da sestavimo degenerirano verigo prostih blokov z enim samim prostim blokom dolžine nič enot (`base`). Tako verigo preiščemo, če je v njej dovolj velik prost blok. Iskati začnemo tam, kjer smo prejšnjikrat nehali. Če najdemo prevelik blok, njegov rep vrnemo kot dodeljen pomnilnik in zmanjšamo dolžino bloka.

Vse dolžine merimo v enotah, ki so `sizeof(union header)` zlogov velike. Manjše enote pomnilnika za nas sploh ne obstajajo. Velikost bloka, ki ga iščemo, je število enot, ki so potrebne, da pokrijejo zahtevano število zlogov plus ena enota za `union header`. Klicatelju vrnemo kazalec na naslednjo enoto. Če dovolj velikega bloka v seznamu ni, dobimo od operacijskega sistema več pomnilnika, ki ga vgradimo v prost seznam, da ga bo `mymalloc` upravljal naprej.

Funkcija `myfree` je načelno preprosta: vrnjeni blok spada med dva obstoječa bloka, ali pa na začetek ali na konec seznama. V vsakem primeru, če se vrnjeni blok dotika svojega levega ali desnega soseda, ga združimo s tem sosedom, tako da nikoli nimamo v prostem seznamu dveh zaporednih prostih blokov, ki se dotikata. Edini problem je skrbeti, da ne zapackamo kazalcev in dolžin.

Funkcija `morecore` poskrbi za dodatni pomnilnik, če v prostem seznamu ni dovolj velikega bloka. Nov blok, ki ga preskrbi sistemska funkcija `sbrk`, je velik najmanj `1024` enot, sicer pa toliko, kot je stranka zahtevala. Teh `1024` enot je zato, da ne bi za vsak mikroskopski blok nadlegovali operacijskega sistema z `sbrk` funkcijo.

Naloga 8.5. Sestavite funkcijo `mycalloc` kot verzijo funkcije `calloc`[12].

---

[12] Pod `UNIX`-om glej `man calloc`

Naloga 8.6. Obstoječa funkcija `myfree` zelo slabo pazi na smiselnost argumenta. Bolj zanesljivo funkcijo dobimo, če `mymalloc` v glavo `union header` doda še kakšno kontrolno informacijo, na primer kazalec nase, `myfree` pa lahko pogleda, če je (še) tam. V `mymalloc` in `myfree` vgradite nekaj takih pasti. Kaj naj `mymalloc/myfree` stori, če odkrije kaj "nezakonitega"?

Naloga 8.7. Sestavite funkcijo `void bfree(void *, size_t)`, ki kot argumenta dobi kazalec na kos statičnega pomnilnika in njegovo velikost, ter ta kos pomnilnika doda v areno, ki jo `myalloc` upravlja.

Dodatek A

Programi

A.1 money.c

```
#ifndef lint
static char *sccsid = "@(#)money.c 1.2";
#endif

/*
 * Money changer
 * Version 1 - no loop, hardwired.
 */

#include <stdio.h>

int
main(void)
{
    int     money;          /* current amount */
    int     n;              /* number of banknotes */

    money = 83456;

    n = money / 10000; money %= 10000;
    (void)printf("%d * %d = %d\n", n, 10000, n * 10000);

    n = money / 5000; money %= 5000;
    (void)printf("%d * %d = %d\n", n, 5000, n * 5000);

    n = money / 1000; money %= 1000;
    (void)printf("%d * %d = %d\n", n, 1000, n * 1000);

    n = money / 500; money %= 500;
    (void)printf("%d * %d = %d\n", n, 500, n * 500);

    n = money / 200; money %= 200;
    (void)printf("%d * %d = %d\n", n, 200, n * 200);

    n = money / 100; money %= 100;
    (void)printf("%d * %d = %d\n", n, 100, n * 100);

    n = money / 50; money %= 50;
    (void)printf("%d * %d = %d\n", n, 50, n * 50);

    n = money / 20; money %= 20;
    (void)printf("%d * %d = %d\n", n, 20, n * 20);

    n = money / 10; money %= 10;
    (void)printf("%d * %d = %d\n", n, 10, n * 10);

    n = money / 5; money %= 5;
    (void)printf("%d * %d = %d\n", n, 5, n * 5);
    n = money / 2; money %= 2;
    (void)printf("%d * %d = %d\n", n, 2, n * 2);

    n = money / 1; money %= 1;
    (void)printf("%d * %d = %d\n", n, 1, n * 1);

    return 0;
}
```

A.2 money2.c

```
#ifndef lint
static char *sccsid = "@(#)money2.c 1.4";
#endif

/*
 * Money changer
 * Version 2 - loop, hardwired.
 */

#include <stdio.h>


int
main(void)
{
    int     money;          /* current amount */
    int     n;              /* number of banknotes */
    int     i;              /* indeks */
    int     banknote[] = {  /* List of slovene banknotes
                             * and coins.  No fractions.
                             */
                10000,
                5000,
                1000,
                500,
                200,
                100,
                50,
                20,
                10,
                5,
                2,
                1,
    };

    money = 83456;

    i = 0;
    while (i < sizeof(banknote) / sizeof(banknote[0])) {
        n = money / banknote[i];
        money = money % banknote[i];
        (void)printf("%d * %d = %d\n",
                     n, banknote[i], n * banknote[i]);
        i = i + 1;
    }

    return 0;
}
```

A.3 money3.c

```
#ifndef lint
static char *sccsid = "@(#)money3.c 1.4";
#endif

/*
 * Money changer
 * Version 3 - loop, hardwired.
 */

#include <stdio.h>

int
main(void)
{
    int     money = 83456;  /* current amount */
    int     n;              /* number of banknotes */
    int     i;              /* indeks */
    int     banknote[] = {  /* List of slovene banknotes
                             * and coins.  No fractions.
                             */
                10000,
                5000,
                1000,
                500,
                200,
                100,
                50,
                20,
                10,
                5,
                2,
                1,
    };

    for (i = 0; i < sizeof(banknote)/sizeof(banknote[0]); ++i) {
        n = money / banknote[i];
        money %= banknote[i];
        if (n > 0)
            (void)printf("%d * %d = %d\n",
                n, banknote[i], n * banknote[i]);
    }

    return 0;
}
```

A.4 money4.c

```
#ifndef lint
static char *sccsid = "@(#)money4.c 1.4";
#endif

/*
 * Money changer
 * Version 4 - loop, hardwired.
 */

#include <stdio.h>

int
main(void)
{
    int     money = 83456;  /* current amount */
    int     n;              /* number of banknotes */
    int     i;              /* indeks */
    int     banknote[] = {  /* List of slovene banknotes
                             * and coins.  No fractions.
                             */
                10000,
                5000,
                1000,
                500,
                200,
                100,
                50,
                20,
                10,
                5,
                2,
                1,
    };

    (void)printf("%d = ", money);
    for (i = 0; i < sizeof(banknote)/sizeof(banknote[0]); ++i) {
        n = money / banknote[i];
        money %= banknote[i];
        if (n > 0)
            (void)printf("%d * %d%s", n, banknote[i],
                money > 0 ? " + " : "\n");
    }

    return 0;
}
```

A.5 myecho.c

```
#ifndef lint
static char *sccsid = "@(#)myecho.c 1.2";
#endif

#include <stdio.h>

int
main(int argc, char *argv[])
{
    int     i;

    for (i = 0; i < argc; ++i)
        (void)printf("argv[%d] = \"%s\"\n", i, argv[i]);

    return 0;
}
```

A.6 money5.c

```
#ifndef lint
static char *sccsid = "@(#)money5.c 1.4";
#endif

/*
 * Money changer
 * Version 5 - loop, command parameter
 */

#include <stdio.h>
#include <stdlib.h>

int
main(int argc, char *argv[])
{
    int     money;          /* current amount */
    int     n;              /* number of banknotes */
    int     i;              /* indeks */
    int     banknote[] = {  /* List of slovene banknotes
                             * and coins.  No fractions.
                             */
                10000,
                5000,
                1000,
                500,
                200,
                100,
                50,
                20,
                10,
                5,
                2,
                1,
    };

    if (argc != 2) {
        (void)fprintf(stderr, "usage: %s <amount>\n", argv[0]);
        return 1;
    }

    money = atoi(argv[1]);
    if (money <= 0) {
        (void)fprintf(stderr, "%s: amount must be positive\n",
            argv[0]);
        return 1;
    }

    (void)printf("%d = ", money);
    for (i = 0; i < sizeof(banknote)/sizeof(banknote[0]); ++i) {
        n = money / banknote[i];
        money %= banknote[i];
        if (n > 0)
            (void)printf("%d * %d%s", n, banknote[i],
                money > 0 ? " + " : "\n");
    }

    return 0;
}
```

A.7 money6.c

```
#ifndef lint
static char *sccsid = "@(#)money6.c 1.6";
#endif

/*
 * Money changer
 * Version 6 - loop, command parameter, floating point
 */

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

int
main(int argc, char *argv[])
{
    double  money;          /* current amount */
    int     n;              /* number of banknotes */
    int     i;              /* indeks */
    double  banknote[] = {  /* List of slovene banknotes
                             * and coins
                             */
                10000,
                5000,
                1000,
                500,
                200,
                100,
                50,
                20,
                10,
                5,
                2,
                1,
                0.50,
    };
    double  small;          /* the smallest banknote (coin) */

    if (argc != 2) {
        (void)fprintf(stderr, "usage: %s <amount>\n", argv[0]);
        return 1;
    }

    money = atof(argv[1]);
    if (money <= 0) {
        (void)fprintf(stderr, "%s: amount must be positive\n",
            argv[0]);
        return 1;
    }

    small = banknote[sizeof(banknote)/sizeof(banknote[0]) - 1];

    (void)printf("%.2f = ", money);

    /* round the amount to small */
    money += small / 2;

    for (i = 0; i < sizeof(banknote)/sizeof(banknote[0]); ++i) {
        n = money / banknote[i];
        money = fmod(money, banknote[i]);
        if (n > 0)
            (void)printf("%d * %.2f%s", n, banknote[i],
                money >= small ? " + " : "\n");
    }

    return 0;
}
```

A.8 mywc.c

```
#ifndef lint
static char *sccsid = "@(#)mywc.c 1.1";
#endif

/*
 * My version of program wc.
 */

#include <stdio.h>
#include <ctype.h>

#define IN      1       /* inside a word */
#define OUT     0       /* outside a word */

int
main(void)
{
    long    nc, nl, nw;
    int     c, state;

    state = OUT;
    nc = nl = nw = 0;
    while ((c = getchar()) != EOF) {
        ++nc;
        if (c == '\n')
            ++nl;
        if (isspace(c))
            state = OUT;
        else if (state == OUT) {
            state = IN;
            ++nw;
        }
    }
    (void)printf("%ld %ld %ld\n", nl, nw, nc);

    return 0;
}
```

A.9 count.c

```
#ifndef lint
static char *sccsid = "@(#)count.c 1.1";
#endif

#include <stdio.h>

/* count digits, white space, and other */

int
main(void)
{
    int     c, i;
    long    nwhite, nother, ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF)
        if (c >= '0' && c <= '9')
            ++ndigit[c - '0'];
        else if (c == ' ' || c == '\n' || c == '\t')
            ++nwhite;
        else
            ++nother;

    (void)printf("digits =");
    for (i = 0; i < 10; ++i)
        (void)printf(" %ld", ndigit[i]);
    (void)printf(", white space = %ld, other = %ld\n",
        nwhite, nother);

    return 0;
}
```

A.10 power.c

```
#ifndef lint
static char *sccsid = "@(#)power.c 1.2";
#endif

#include <stdio.h>

static int power(int base, int n);

/* test power function */
int
main(void)
{
    int     i;

    for (i = 0; i < 10; ++i)
        (void)printf("%d %d %d\n", i, power(2,i), power(-3,i));

    return 0;
}

/* power: raise base to n-th power; n >= 0 */
static int
power(int base, int n)
{
    int     i, p;

    p = 1;
    for (i = 1; i <= n; ++i)
        p *= base;

    return p;
}
```

A.11 longest.c

```
#ifndef lint
static char *sccsid = "@(#)longest.c 1.2";
#endif

#include <stdio.h>

#define MAXLINE 1000    /* maximum input line size */

static int getline(char line[], int maxline);
static void copy(char to[], char from[]);

/* print longest input line */
int
main(void)
{
    int     len;                /* current line length */
    int     max;                /* maximum length seen so far */
    char    line[MAXLINE];      /* current input line */
    char    longest[MAXLINE];   /* longest line saved here */

    max = 0;
    while ((len = getline(line, MAXLINE)) > 0)
        if (len > max) {
            max = len;
            copy(longest, line);
        }

    if (max > 0)    /* there was a line */
        (void)printf("%s", longest);

    return 0;
}


/* getline: read a line of input into s, return length */
static int
getline(char s[], int lim)
{
    int     c, i;

    for (i = 0;
         i < lim - 1 && (c = getchar()) != EOF && c != '\n';
         ++i)
        s[i] = (char)c;
    if (c == '\n')
        s[i++] = (char)c;
    s[i] = '\0';
    return i;
}


/* copy: copy 'from' into 'to'; assume 'to' is big enough */
static void
copy(char to[], char from[])
{
    int     i;

    for (i = 0; (to[i] = from[i]) != '\0'; ++i)
        ;
}
```

A.12 binsearch.c

```
#ifndef lint
static char *sccsid = "@(#)binsearch.c 1.2";
#endif

#include "proto.h"

/* binsearch: find x in v[0] <= v[1] <= ... <= v[n-1] */
int
binsearch(const int x, const int v[], const int n)
{
    int     low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if (x < v[mid])
            high = mid - 1;
        else if (x > v[mid])
            low = mid + 1;
        else
            return mid;
    }
    return -1;              /* Not found */
}
```

A.13 proto.h

```
/* @(#)proto.h 1.1*/

int binsearch(const int x, const int v[], const int n);
int myatoi(const char s[]);
void shellsort(int v[], const int n);
void reverse(char v[]);
void itoa(int n, char s[]);
```

A.14 count2.c

```
#ifndef lint
static char *sccsid = "@(#)count.c 1.1";
#endif

#include <stdio.h>

/* count digits, white space, and other */

int
main(void)
{
    int     c, i;
    long    nwhite, nother, ndigit[10];

    nwhite = nother = 0;
    for (i = 0; i < 10; ++i)
        ndigit[i] = 0;

    while ((c = getchar()) != EOF) {
        switch (c) {
        case '0': case '1': case '2': case '3': case '4':
        case '5': case '6': case '7': case '8': case '9':
            ++ndigit[c - '0'];
            break;
        case ' ':
        case '\n':
        case '\t':
            ++nwhite;
            break;
        default:
            ++nother;
            break;
        }
    }

    (void)printf("digits =");
    for (i = 0; i < 10; ++i)
        (void)printf(" %ld", ndigit[i]);
    (void)printf(", white space = %ld, other = %ld\n",
        nwhite, nother);

    return 0;
}
```

A.15 myatoi.c

```
#ifndef lint
static char *sccsid = "@(#)myatoi.c 1.2";
#endif

#include <ctype.h>
#include "proto.h"

/* myatoi: convert s to integer; version 2 */
int
myatoi(const char s[])
{
    int     i, n, negative;

    for (i = 0; isspace(s[i]); ++i)
        ;       /* skip white space */

    negative = s[i] == '-';
    if (s[i] == '+' || s[i] == '-')
        ++i;    /* skip sign */

    for (n = 0; isdigit(s[i]); ++i)
        n = 10 * n + (s[i] - '0');

    return negative ? -n : +n;
}
```

A.16 shellsort.c

```
#ifndef lint
static char *sccsid = "@(#)shellsort.c 1.2";
#endif

#include "proto.h"

/* shellsort: sort v[0] ... v[n-1] into increasing order */
void
shellsort(int v[], const int n)
{
    int     gap, i, j, temp;

    for (gap = n/2; gap > 0; gap /= 2)
        for (i = gap; i < n; ++i)
            for (j = i - gap;
                 j >= 0 && v[j] > v[j + gap];
                 j -= gap) {
                temp = v[j];
                v[j] = v[j + gap];
                v[j + gap] = temp;
            }
}
```

A.17 reverse.c

```
#ifndef lint
static char *sccsid = "@(#)reverse.c 1.2";
#endif

#include <string.h>
#include "proto.h"

/* reverse: reverse string s in place */
void
reverse(char s[])
{
    int     i, j;
    char    c;

    for (i = 0, j = strlen(s)-1; i < j; ++i, --j)
        c = s[i], s[i] = s[j], s[j] = c;
}
```

A.18 itoa.c

```
#ifndef lint
static char *sccsid = "@(#)itoa.c 1.2";
#endif

#include "proto.h"

/* itoa: convert n to characters in s */
void
itoa(int n, char s[])
{
    int     i, sign;

    if ((sign = n) < 0)         /* record sign */
        n = -n;

    i = 0;
    do {
        s[i++] = n % 10 + '0';  /* get next digit */
    } while ((n /= 10) > 0);    /* delete it */
    if (sign < 0)
        s[i++] = '-';
    s[i] = '\0';
    reverse(s);
}
```

A.19 mygrep.c

```
#ifndef lint
static char *sccsid = "@(#)mygrep.c 1.2";
#endif

#include <stdio.h>
#define MAXLINE 1000        /* maximum input line length */

static int getline(char line[], int max);
static int strindex(const char source[], const char searchfor[]);

static char pattern[] = "ould";

/* find all lines matching pattern */
int
main(void)
{
    char    line[MAXLINE];
    int     found;

    found = 0;
    while (getline(line, MAXLINE) > 0)
        if (strindex(line, pattern) >= 0) {
            (void)printf("%s", line);
            ++found;
        }
    return found;
}

/* getline: get line into s, return length */
static int
getline(char s[], int lim)
{
    int     c, i;

    i = 0;
    while (--lim > 0 && (c = getchar()) != EOF && c != '\n')
        s[i++] = (char)c;
    if (c == '\n')
        s[i++] = (char)c;
    s[i] = '\0';

    return i;
}

/* strindex: return index of t in s, -1 if none */
static int
strindex(const char s[], const char t[])
{
    int      i, j, k;

    for (i = 0; s[i] != '\0'; ++i) {
        for (j = i, k = 0;
             t[k] != '\0' && s[j] == t[k];
             ++j, ++k) ;
        if (k > 0 && t[k] == '\0')
            return i;
    }
    return -1;
}
```

A.20 myatof.c

```
#ifndef lint
static char *sccsid = "@(#)myatof.c 1.2";
#endif

#include <ctype.h>

/* myatof: convert string s to double */
double
myatof(const char s[])
{
    double  val, power;
    int     i, sign;

    for (i = 0; isspace(s[i]); ++i)
        ;                   /* skip white space */
    sign = (s[i] == '-') ? -1 : +1;
    if (s[i] == '+' || s[i] == '-')
        ++i;
    for (val = 0.0; isdigit(s[i]); ++i)
        val = 10.0 * val + (s[i] - '0');
    if (s[i] == '.')
        ++i;
    for (power = 1.0; isdigit(s[i]); ++i) {
        val = 10.0 * val + (s[i] - '0');
        power *= 10.0;
    }
    return sign * val / power;
}
```

A.21 xcalc.c

```
#ifndef lint
static char *sccsid = "@(#)xcalc.c 1.3";
#endif

#include <stdio.h>
#include "proto4.h"

#define MAXLINE 100

/* rudimentary calculator */
int
main(void)
{
    double  sum;
    char    line[MAXLINE];

    sum = 0;
    while (getline(line, MAXLINE) > 1)
        (void)printf("\t%g\n", sum += myatof(line));

    return 0;
}
```

A.22 calc.c

```
#ifndef lint
static char *sccsid = "@(#)calc.c 1.2";
#endif

#include <stdio.h>
#include <math.h>       /* for atof() */

#define MAXOP   100     /* max size of operand
                         * or operator string
                         */
#define NUMBER  '0'     /* signal that a number
                         * was found
                         */

static int getop(char[]);
static void push(double);
static double pop(void);

/* reverse Polish calculator */
int
main(void)
{
    int     type;
    double  op2;
    char    s[MAXOP];

    while ((type = getop(s)) != EOF) {
        switch (type) {
        case NUMBER:
            push(atof(s));
            break;
        case '+':
            push(pop() + pop());
            break;
        case '*':
            push(pop() * pop());
            break;
        case '-':
            op2 = pop();
            push(pop() - op2);
            break;
        case '/':
            op2 = pop();
            push(pop() / op2);
            break;
        case '\n':
            (void)printf("\t%.8g\n", pop());
            break;
        default:
            (void)fprintf(stderr,
                "error: unknown command %s\n", s);
            break;
        }
    }
    return 0;
}
```

```
#define MAXVAL  100             /* maximum depth of val stack */

static int sp = 0;              /* next free stack position */
static double val[MAXVAL];      /* value stack */

/* push: push f onto value stack */
static void
push(double f)
{
    if (sp < MAXVAL)
        val[sp++] = f;
    else
        (void)fprintf(stderr,
            "error:stack full, can't push %g\n", f);
}

/* pop: pop and return top value from stack */
static double
pop(void)
{
    if (sp > 0)
        return val[--sp];
    else {
        (void)fprintf(stderr, "error: stack empty\n");
        return 0.0;
    }
}
```

```
#include <ctype.h>

static int getch(void);
static void ungetch(int);

/* getop: get next operator or numeric operand */
static int
getop(char s[])
```

```
{
    int     i, c;

    while ((s[0] = c = getch()) == ' ' || c == '\t')
        ;
    if (!isdigit(c) && c != '.')
        return c;       /* not a number */
    i = 0;
    if (isdigit(c))     /* collect integer part */
        while (isdigit(s[++i] = c = getch()))
            ;
    if (c == '.')       /* collect fraction part */
        while (isdigit(s[++i] = c = getch()))
            ;
    s[i] = '\0';
    if (c != EOF)
        ungetch(c);
    return NUMBER;
}



#define BUFSIZE 100

static char buf[BUFSIZE];       /* buffer for ungetch */
static int  bufp = 0;           /* next free position in buf */

/* getch: get a (possibly pushed back) character */
static int
getch(void)
{
    return (bufp > 0) ? buf[--bufp] : getchar();
}

/* ungetch: push character c back on input */
static void
ungetch(int c)
{
    if (bufp >= BUFSIZE)
        (void)fprintf(stderr, "ungetch: too many characters\n");
    else
        buf[bufp++] = (char)c;
}
```

A.23 qsort.c

```
#ifndef lint
static char *sccsid = "@(#)qsort.c 1.3";
#endif

#include "proto4.h"

static void swap(int [], int, int);

/* myqsort: sort v[left] ... v[right] into increasing order */
void
myqsort(int v[], int left, int right)
{
    int     i, last;

    if (left >= right)      /* do nothing if array contains */
        return;             /* fewer than two elements */
    swap(v, left, (left + right)/2);/* move partition element */
    last = left;                    /* to v[0] */
    for (i = left + 1; i <= right;  ++i)    /* partition */
        if (v[i] < v[left])
            swap(v, ++last, i);
    swap(v, left, last);    /* restore partition element */
    myqsort(v, left, last - 1);
    myqsort(v, last + 1, right);
}

static void
swap(int v[], int i, int j)
{
    int     temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

A.24 proto4.h

```
/* @(#)proto4.h 1.1 */

int getline(char s[], int max);
double myatof(const char s[]);
void myqsort(int v[], int i, int j);
```

A.25 fib.c

```
#ifndef lint
static char *sccsid = "@(#)fib.c 1.4";
#endif

#include <stdio.h>
#include <stdlib.h>

static int lfib(int);
static int rfib(int);

/* Fibonacci numbers
 *
 *     fib [r|l] num
 */
int
main(int argc, char *argv[])
{
    char    *type;
    int     num;

    if (argc != 3) {
        (void)fprintf(stderr, "usage: %s [r|l] num\n", argv[0]);
        return 1;
    }

    type = argv[1];
    num = atoi(argv[2]);

    (void)printf("fib[%d] = %d\n", num,
        (*type == 'l' ? lfib : rfib)(num));

    return 0;
}

/* Fibonacci using loop */
static int
lfib(int n)
{
    int     f0, f1, f2;

    f0 = f1 = 1;
    switch (n) {
    case 0:
        return f0;
    case 1:
        return f1;
    default:
        while (n > 1) {
            f2 = f0 + f1;
            f0 = f1;
            f1 = f2;
            --n;
        }
        return f2;
    }
}

/* Fibonacci using recursion */
static int
rfib(int n)
{
    switch (n) {
    case 0:
    case 1:
        return 1;
    default:
        return rfib(n-1) + rfib(n-2);
    }
}
```

A.26 alloc.c

```
#ifndef lint
static char *sccsid = "@(#)alloc.c 1.2";
#endif

#include <stdlib.h>

#define ALLOCSIZE   10000           /* size of available space */

static char allocbuf[ALLOCSIZE];    /* storage for alloc */
static char *allocp = allocbuf;     /* next free position */

void *
alloc(size_t size)                  /* allocate size bytes */
{
    if (&allocp[size] > &allocbuf[ALLOCSIZE])
        return NULL;                /* not enough room */

    allocp += size;                 /* allocate size bytes */
    return allocp - size;           /* return old allocp */
}

void
afree(void *p)              /* free storage pointed to by p */
{
    if (p >= (void *)allocbuf &&
        p < (void *)&allocbuf[ALLOCSIZE])
        allocp = p;
}
```

A.27 sortlines.c

```
#ifndef lint
static char *sccsid = "@(#)sortlines.c 1.4";
#endif

#include <stdio.h>
#include <string.h>

#define MAXLINES    5000        /* max #lines to be sorted */

static char *lineptr[MAXLINES]; /* pointers to text lines */

static void swap(char *[], const int, const int);

extern void *alloc(size_t);
extern void afree(void *);

static int readlines(char *[], const int);
static void freelines(char *[], const int);
static void writelines(char *[], const int);
static void myqsort(char *[], const int, const int);

/* sort input lines */
int
main(void)
{
    int     nl;     /* number of input lines read */

    if ((nl = readlines(lineptr, MAXLINES)) >= 0) {
        myqsort(lineptr, 0, nl - 1);
        writelines(lineptr, nl);
        freelines(lineptr, nl);
        return 0;
    }
    (void)fprintf(stderr, "error: input too big for sort\n");
    return 1;
}


#define MAXLEN  1000        /* max length of any input line */

int getline(char *, int);

/* readlines: read input lines */
static int
readlines(char *lineptr[], const int maxlines)
{
    int     len, nl;
    char    *p, line[MAXLEN];
    for (nl = 0; (len = getline(line, MAXLEN)) > 0; ++nl) {
        if (nl >= maxlines || (p = alloc(len)) == NULL)
            return -1;
        line[len-1] = '\0';     /* delete newline */
        (void)strcpy(p, line);
        lineptr[nl] = p;
    }
    return nl;
}

/* writelines: write output lines */
static void
writelines(char *lineptr[], const int nl)
{
    int     i;

    for (i = 0; i < nl; ++i)
        (void)printf("%s\n", lineptr[i]);
}

/* freelines: free lines */
static void
freelines(char *lineptr[], const int nl)
{
    int     i;

    for (i = nl - 1; i >= 0; --i)
        afree(lineptr[i]);
}


/* myqsort:
 * sort v[left] ... v[right] into increasing order
 */
static void
myqsort(char *v[], const int left, const int right)
{
    int     i, last;

    if (left >= right)      /* do nothing if array contains */
        return;             /* fewer than two elements */
    swap(v, left, (left + right)/2);
                            /* move partition element */
    last = left;                            /* to v[0] */
    for (i = left + 1; i <= right;  ++i)    /* partition */
        if (strcmp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);    /* restore partition element */
    myqsort(v, left, last - 1);
    myqsort(v, last + 1, right);
}

static void
swap(char *v[], const int i, const int j)
{
    char    *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}
```

A.28 dayofyear.c

```
#ifndef lint
static char *sccsid = "@(#)dayofyear.c 1.2";
#endif

#include <stdio.h>

static int day_of_year(int, int, int);
static void month_day(int, int, int *, int *);

int
main(void)
{
    int     y, m, d, j;

    while (scanf("%d %d %d", &y, &m, &d) == 3) {
        (void)printf("%d, %d, %d, %d\n",
                     y, m, d, j = day_of_year(y, m, d));
        m = d = 0;
        month_day(y, j, &m, &d);
        (void)printf("%d, %d, %d, %d\n", y, j, m, d);
    }
    return 0;
}


static char daytab[2][13] = {
    {0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
    {0, 31, 29, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31},
};

/* day_of_year: set day of year from month and day */
static int
day_of_year(int year, int month, int day)
{
    int     i, leap;

    leap = year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
    for (i = 1; i < month; ++i)
        day += daytab[leap][i];

    return day;
}

/* month_day: set month, day from day of year */
static void
month_day(int year, int yearday, int *pmonth, int *pday)
{
    int     i, leap;
    leap = year % 4 == 0 && year % 100 != 0 || year % 400 == 0;
    for (i = 1; yearday > daytab[leap][i]; ++i)
        yearday -= daytab[leap][i];
    *pmonth = i;
    *pday = yearday;
}
```

A.29 mygrep1.c

```
#ifndef lint
static char *sccsid = "@(#)mygrep1.c 1.1";
#endif

#include <stdio.h>
#include <string.h>

#define MAXLINE 1000        /* maximum input line length */

static int getline(char line[], int max);

/* mygrep1: print all lines matching pattern from 1st arg */
int
main(int argc, char *argv[])
{
    char    line[MAXLINE];
    int     found = 0;

    if (argc != 2)
        (void)fprintf(stderr, "Usage: %s pattern\n", argv[0]);
    else
        while (getline(line, MAXLINE) > 0)
            if (strstr(line, argv[1]) != NULL) {
                (void)printf("%s", line);
                ++found;
            }
    return found;
}

/* getline: get line into s, return length */
static int
getline(char s[], int lim)
{
    int     c, i;

    i = 0;
    while (--lim > 0 && (c = getchar()) != EOF && c != '\n')
        s[i++] = (char)c;
    if (c == '\n')
        s[i++] = (char)c;
    s[i] = '\0';

    return i;
}
```

A.30 mygrep2.c

```
#ifndef lint
static char *sccsid = "@(#)mygrep2.c 1.3";
#endif

#include <stdio.h>
#include <string.h>

#define MAXLINE 1000        /* maximum input line length */

static int getline(char line[], int max);

/* mygrep2: print all lines matching pattern from 1st arg */
int
main(int argc, char *argv[])
{
    char    line[MAXLINE];
    long    lineno = 0;
    int     c, except = 0, number = 0, found = 0;
    char    *pname = argv[0];

    while (--argc > 0 && (*++argv)[0] == '-')
        while ((c = *++argv[0]) != '\0')
            switch (c) {
            case 'x':
                except = 1;
                break;
            case 'n':
                number = 1;
                break;
            default:
                (void)fprintf(stderr, "%s: illegal option %c\n",
                    pname, c);
                argc = 0;
                found = -1;
                break;
            }

    if (argc != 1)
        (void)fprintf(stderr, "Usage: %s [-x] [-n] pattern\n",
            pname);
    else
        while (getline(line, MAXLINE) > 0) {
            ++lineno;
            if ((strstr(line, *argv) != NULL) != except) {
                if (number)
                    (void)printf("%ld", lineno);
                (void)printf("%s", line);
                ++found;
            }
        }
    return found;
}

/* getline: get line into s, return length */
static int
getline(char s[], int lim)
{
    int     c, i;

    i = 0;
    while (--lim > 0 && (c = getchar()) != EOF && c != '\n')
        s[i++] = (char)c;
    if (c == '\n')
        s[i++] = (char)c;
    s[i] = '\0';

    return i;
}
```

A.31 sortlines2.c

```
#ifndef lint
static char *sccsid = "@(#)sortlines2.c 1.4";
#endif

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define MAXLINES    5000        /* max #lines to be sorted */

static char *lineptr[MAXLINES]; /* pointers to text lines */

static int readlines(char *[], int);
static void writelines(char *[], int);
static void myqsort(void *[], int, int, int (*)(void *, void *));
static void swap(void *[], int, int);
static int numcmp(const char *, const char *);

/* sort input lines, version 2 */
int
main(int argc, char *argv[])
{
    int     nlines;         /* number of input lines read */
    int     numeric = 0;    /* 1 if numeric sort */

    if (argc > 1 && strcmp(argv[1], "-n") == 0)
        numeric = 1;
    if ((nlines = readlines(lineptr, MAXLINES)) >= 0) {
        myqsort((void **)lineptr, 0, nlines - 1,
          (int (*)(void *, void *)) (numeric ? numcmp : strcmp));
        writelines(lineptr, nlines);
        return 0;
    }
    (void)fprintf(stderr, "error: input too big for sort\n");
    return 1;
}


#define MAXLEN  1000        /* max length of any input line */

int getline(char *, int);

/* readlines: read input lines */
static int
readlines(char *lineptr[], int maxlines)
{
    int     len, nlines;
    char    *p, line[MAXLEN];
    for (nlines = 0;
           (len = getline(line, MAXLEN)) > 0; ++nlines) {
        if (nlines >= maxlines || (p = malloc(len)) == NULL)
            return -1;
        line[len-1] = '\0';     /* delete newline */
        (void)strcpy(p, line);
        lineptr[nlines] = p;
    }
    return nlines;
}

/* writelines: write output lines */
static void
writelines(char *lineptr[], int nlines)
{
    int     i;

    for (i = 0; i < nlines; ++i)
        (void)printf("%s\n", lineptr[i]);
}


/* myqsort: sort v[left] ... v[right] into increasing order */
static void
myqsort(void *v[], int left, int right,
      int (*comp)(void *, void*))
{
    int     i, last;

    if (left >= right)      /* do nothing if array contains */
        return;             /* fewer than two elements */
    swap(v, left, (left + right)/2);
                            /* move partition element */
    last = left;            /* to v[0] */
    for (i = left + 1; i <= right;  ++i)  /* partition */
        if (comp(v[i], v[left]) < 0)
            swap(v, ++last, i);
    swap(v, left, last);    /* restore partition element */
    myqsort(v, left, last - 1, comp);
    myqsort(v, last + 1, right, comp);
}

static void
swap(void *v[], int i, int j)
{
    void    *temp;

    temp = v[i];
    v[i] = v[j];
    v[j] = temp;
}

static int
numcmp(const char *s1, const char *s2)
{
    double      v1, v2;

    v1 = atof(s1);
    v2 = atof(s2);
    if (v1 < v2)
        return -1;
    if (v1 > v2)
        return +1;
    return 0;
}
```

A.32 matrix.c

```
#ifndef lint
static char *sccsid = "@(#)matrix.c 1.3";
#endif

#include <stdio.h>
#include <stdlib.h>

static double **dmatrix(const long, const long);

/* test dmatrix */
int
main(void)
{
    double  **a;

    if ((a = dmatrix(3, 4)) == NULL) {
        (void)printf("no memory\n");
        return 1;
    }

    free(a);
    return 0;
}

/* dmatrix: allocate double matrix of order m * n */
static double **
dmatrix(const long m, const long n)
{
    double  **p, *q;
    int     i;

    p = malloc(m * sizeof(double *) + m * n * sizeof(double));
    if (p != NULL)
        for (i = 0, q = (double *)&p[m]; i < m; ++i, q += n)
            p[i] = q;

    return p;
}
```

A.33 dcl.c

```
#ifndef lint
static char *sccsid = "@(#)dcl.c 1.2";
#endif

#include <stdio.h>
#include <string.h>
#include <ctype.h>

#define MAXTOKEN    100

enum { NAME, PARENS, BRACKETS };

static void dcl(void);
static void dirdcl(void);

static int gettoken(void);
static int tokentype;           /* type of last token */
static char token[MAXTOKEN];    /* last token string */
static char name[MAXTOKEN];     /* identifier name */
static char datatype[MAXTOKEN];
static char out[1000];          /* output string */

int
main(void)
{
    while (gettoken() != EOF) {         /* first token on line */
        (void)strcpy(datatype, token);  /* is the datatype */
        out[0] = '\0';
        dcl();                          /* parse rest of line */
        if (tokentype != '\n')
            (void)printf("syntax error\n");
        (void)printf("%s: %s %s\n", name, out, datatype);
    }
    return 0;
}

static int
gettoken(void)
{
    int     c;
    char    *p = token;

    while ((c = getchar()) == ' ' || c == '\t')
        ;
    if (c == '(') {
        if ((c = getchar()) == ')') {
            (void)strcpy(token, "()");
            tokentype = PARENS;
        }
        else {
            (void)ungetc(c, stdin);
            tokentype = '(';
        }
    }
    else if (c == '[') {
        for (*p++ = (char)c; (*p++ = (char)getchar()) != ']'; )
            ;
        *p = '\0';
        tokentype = BRACKETS;
    }
    else if (isalpha(c)) {
        for (*p++ = (char)c; isalnum(c = getchar()); )
            *p++ = (char)c;
        *p = '\0';
        (void)ungetc(c, stdin);
        tokentype = NAME;
    }
    else
        tokentype = c;
    return tokentype;
}


/* dcl: parse a declarator */
void
dcl(void)
{
    int     ns;

    for (ns = 0; gettoken() == '*'; ++ns)       /* count *s */
        ;
    dirdcl();
    while (ns-- > 0)
        (void)strcat(out, " pointer to");
}

/* dirdcl: parse a direct declarator */
void
dirdcl(void)
{
    int     type;

    if (tokentype == '(') {         /* ( dcl ) */
        dcl();
        if (tokentype != ')')
            (void)printf("error: missing )\n");
    }
    else if (tokentype == NAME)     /* variable name */
```

```
        (void)strcpy(name, token);
    else
        (void)printf("error: expected name or (dcl)\n");

    while ((type = gettoken()) == PARENS || type == BRACKETS) {
        if (type == PARENS)
            (void)strcat(out, " function returning");
        else {
            (void)strcat(out, " array");
            (void)strcat(out, token);
            (void)strcat(out, " of");
        }
    }
}
```

A.34 fract.c

```
#ifndef lint
static char *sccsid = "@(#)fract.c 1.7";
#endif

/* fraction arithmetics */

#include <stdio.h>
#include <stdlib.h>

struct fraction {
    int     num;    /* numerator */
    int     den;    /* denominator */
};

static struct fraction makefraction(const int, const int);
static struct fraction addfraction(struct fraction,
                                   struct fraction);
static struct fraction canonfract(struct fraction);
static int GCD(int, int);



/* add two fractions */
int
main(int argc, char *argv[])
{
    struct fraction f, g, s;

    if (argc != 5) {
        (void)fprintf(stderr, "usage: %s n1 d1 n2 d2\n",
            argv[0]);
        return 1;
    }

    f = makefraction(atoi(argv[1]), atoi(argv[2]));
    g = makefraction(atoi(argv[3]), atoi(argv[4]));
    s = addfraction(f, g);
    (void)printf("%d/%d + %d/%d = %d/%d\n",
        f.num, f.den, g.num, g.den, s.num, s.den);

    return 0;
}


/* make a fraction from num, den components */
static struct fraction
makefraction(const int num, const int den)
{
    struct fraction temp;

    temp.num = num;
    temp.den = den;
    return temp;
}


/* add fractions p and q */
static struct fraction
addfraction(struct fraction p, struct fraction q)
{
    struct fraction s;
    int             gcd;

    p = canonfract(p);
    q = canonfract(q);
    gcd = GCD(p.den, q.den);
    s.den = p.den / gcd * q.den;
    s.num = q.den / gcd * p.num + p.den / gcd * q.num;
    return canonfract(s);
}


/* return canonical form of fraction f */
static struct fraction
canonfract(struct fraction f)
{
    int gcd;

    if (f.den == 0) {
        (void)fprintf(stderr, "canonfract: zero denominator\n");
        exit(1);
    }
    gcd = GCD(f.num, f.den);
    f.num /= gcd;
    f.den /= gcd;
    if (f.den < 0)
        f.num = -f.num, f.den = -f.den;
    return f;
}


/* return greatest common divisor of a and b */
static int
GCD(int a, int b)
{
    int     r;

    a = abs(a);
    b = abs(b);
    while (b > 0) {
        r = a % b, a = b, b = r;
    }
    return a;
}
```

A.35 keyword1.c

```
#ifndef lint
static char *sccsid = "@(#)keyword1.c 1.2";
#endif

/* keyword: keyword counting program - array version */

static struct key {
    char    *word;
    int     count;
} keytab[] = {
    { "auto",       0 },
    { "break",      0 },
    { "case",       0 },
    { "char",       0 },
    { "const",      0 },
    { "continue",   0 },
    { "default",    0 },
    { "do",         0 },
    { "double",     0 },
    { "else",       0 },
    { "enum",       0 },
    { "extern",     0 },
    { "float",      0 },
    { "for",        0 },
    { "goto",       0 },
    { "if",         0 },
    { "int",        0 },
    { "long",       0 },
    { "register",   0 },
    { "return",     0 },
    { "short",      0 },
    { "signed",     0 },
    { "sizeof",     0 },
    { "static",     0 },
    { "struct",     0 },
    { "switch",     0 },
    { "typedef",    0 },
    { "union",      0 },
    { "unsigned",   0 },
    { "void",       0 },
    { "volatile",   0 },
    { "while",      0 },
};

#define NKEYS   (sizeof(keytab) / sizeof(keytab[0]))

#include <stdio.h>
#include <ctype.h>
```

```
#include <string.h>

#define MAXWORD 100

static int getword(char *, int);
static int binsearch(char *, struct key *, int);

/* count C keywords */
int
main(void)
{
    int     n;
    char    word[MAXWORD];

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((n = binsearch(word, keytab, NKEYS)) >= 0)
                ++keytab[n].count;

    for (n = 0; n < NKEYS; ++n)
        if (keytab[n].count > 0)
            (void)printf("%4d %s\n",
                keytab[n].count, keytab[n].word);

    return 0;
}


/* binsearch: find word in tab[0] ... tab[n-1] */
static int
binsearch(char *word, struct key tab[], int n)
{
    int     cond;
    int     low, high, mid;

    low = 0;
    high = n - 1;
    while (low <= high) {
        mid = (low + high) / 2;
        if ((cond = strcmp(word, tab[mid].word)) < 0)
            high = mid - 1;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }

    return -1;
}
```

```
static int
getword(char *word, int lim)
{
    int     c;
    char    *w = word;

    while (isspace(c = getchar()))
        ;

    if (c != EOF)
        *w++ = (char)c;
    if (!isalpha(c)) {
        *w = '\0';
        return c;
    }
    for (; --lim > 0; ++w)
        if (!isalnum(*w = (char)getchar())) {
            (void)ungetc(*w, stdin);
            break;
        }
    *w = '\0';
    return word[0];
}
```

A.36 keyword2.c

```
#ifndef lint
static char *sccsid = "@(#)keyword2.c 1.2";
#endif

/* keyword: keyword counting program - pointer version */

static struct key {
    char    *word;
    int     count;
} keytab[] = {
    { "auto",       0 },
    { "break",      0 },
    { "case",       0 },
    { "char",       0 },
    { "const",      0 },
    { "continue",   0 },
    { "default",    0 },
    { "do",         0 },
    { "double",     0 },
    { "else",       0 },
    { "enum",       0 },
    { "extern",     0 },
    { "float",      0 },
    { "for",        0 },
    { "goto",       0 },
    { "if",         0 },
    { "int",        0 },
    { "long",       0 },
    { "register",   0 },
    { "return",     0 },
    { "short",      0 },
    { "signed",     0 },
    { "sizeof",     0 },
    { "static",     0 },
    { "struct",     0 },
    { "switch",     0 },
    { "typedef",    0 },
    { "union",      0 },
    { "unsigned",   0 },
    { "void",       0 },
    { "volatile",   0 },
    { "while",      0 },
};

#define NKEYS   (sizeof(keytab) / sizeof(keytab[0]))

#include <stdio.h>
#include <ctype.h>
```

```
#include <string.h>

#define MAXWORD 100

static int getword(char *, int);
static struct key *binsearch(char *, struct key *, int);

/* count C keywords */
int
main(void)
{
    char        word[MAXWORD];
    struct key  *p;

    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            if ((p = binsearch(word, keytab, NKEYS)) != NULL)
                ++p->count;

    for (p = keytab; p < &keytab[NKEYS]; ++p)
        if (p->count > 0)
            (void)printf("%4d %s\n", p->count, p->word);

    return 0;
}


/* binsearch: find word in tab[0] ... tab[n-1] */
static struct key *
binsearch(char *word, struct key tab[], int n)
{
    int     cond;
    struct key  *low = &tab[0];
    struct key  *high = &tab[n];
    struct key  *mid;

    while (low < high) {
        mid = low + (high - low) / 2;
        if ((cond = strcmp(word, mid->word)) < 0)
            high = mid;
        else if (cond > 0)
            low = mid + 1;
        else
            return mid;
    }

    return NULL;
}

static int
```

```
getword(char *word, int lim)
{
    int     c;
    char    *w = word;

    while (isspace(c = getchar()))
        ;

    if (c != EOF)
        *w++ = (char)c;
    if (!isalpha(c)) {
        *w = '\0';
        return c;
    }
    for (; --lim > 0; ++w)
        if (!isalnum(*w = (char)getchar())) {
            (void)ungetc(*w, stdin);
            break;
        }
    *w = '\0';
    return word[0];
}
```

A.37 freq.c

```
#ifndef lint
static char *sccsid = "@(#)freq.c 1.3";
#endif

/* freq: frequency of words */


struct tnode {              /* the tree node */
    char            *word;  /* points to the text */
    int             count;  /* number of occurences */
    struct tnode    *left;  /* left child */
    struct tnode    *right; /* right child */
};

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>

#define MAXWORD 100

static int getword(char *, int);
static struct tnode *addtree(struct tnode *, char *);
static void treeprint(const struct tnode *);

/* word frequency count */
int
main(void)
{
    struct tnode    *root;
    char            word[MAXWORD];

    root = NULL;
    while (getword(word, MAXWORD) != EOF)
        if (isalpha(word[0]))
            root = addtree(root, word);
    treeprint(root);

    return 0;
}


/* addtree: add a node with w, at or below p */
static struct tnode *
addtree(struct tnode *p, char *w)
{
    int     cond;
    if (p == NULL) {    /* new word has arrived */
        if ((p = malloc(sizeof(struct tnode))) != NULL) {
            p->word = strdup(w);
            p->count = 1;
            p->left = p->right = NULL;
        }
    }
    else if ((cond = strcmp(w, p->word)) == 0)
        ++p->count;     /* repeated word */
    else if (cond < 0)  /* less than into left subtree */
        p->left = addtree(p->left, w);
    else                /* greater than into right subtree */
        p->right = addtree(p->right, w);

    return p;
}

static int
getword(char *word, int lim)
{
    int     c;
    char    *w = word;

    while (isspace(c = getchar()))
        ;

    if (c != EOF)
        *w++ = (char)c;
    if (!isalpha(c)) {
        *w = '\0';
        return c;
    }
    for (; --lim > 0; ++w)
        if (!isalnum(*w = (char)getchar())) {
            (void)ungetc(*w, stdin);
            break;
        }
    *w = '\0';
    return word[0];
}


/* treeprint: in-order print of tree */
static void
treeprint(const struct tnode *p)
{
    if (p != NULL) {
        treeprint(p->left);
        (void)printf("%4d %s\n", p->count, p->word);
        treeprint(p->right);
    }
}
```

A.38 macro.c

```
#ifndef lint
static char *sccsid = "@(#)macro.c 1.2";
#endif

/* macro: test program for macro package */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>


struct nlist {
    struct nlist    *next;      /* next entry in the chain */
    char            *name;      /* defined name */
    char            *defn;      /* replacement text */
};

#define HASHSIZE    101
static struct nlist *hashtab[HASHSIZE];     /* pointer table */


static struct nlist *lookup(const char *);
static struct nlist *install(const char *, const char *);
static unsigned hash(const char *);


int
main(void)
{
    struct nlist    *np;

    if (install("IN", "0") == NULL ||
        install("OUT", "1") == NULL) {
        (void)fprintf(stderr, "%s\n", "Can't install");
        return 1;
    }

    if ((np = lookup("IN")) == NULL) {
        (void)fprintf(stderr, "%s\n", "Can't lookup");
        return 1;
    }
    (void)printf("\"%s\" = \"%s\"\n", np->name, np->defn);

    if ((np = lookup("OUT")) == NULL) {
        (void)fprintf(stderr, "%s\n", "Can't lookup");
        return 1;
    }
    (void)printf("\"%s\" = \"%s\"\n", np->name, np->defn);
    return 0;
}




/* hash: form hash value for string s */
static unsigned
hash(const char *s)
{
    unsigned    hashval;

    for (hashval = 0; *s; ++s)
        hashval = *s + 31 * hashval;

    return hashval % HASHSIZE;
}


/* lookup: look for s in hashtab */
static struct nlist *
lookup(const char *s)
{
    struct nlist    *np;

    for (np = hashtab[hash(s)]; np != NULL; np = np->next)
        if (strcmp(s, np->name) == 0)
            return np;
    return NULL;
}


/* install: put (name, defn) in hashtab */
static struct nlist *
install(const char *name, const char *defn)
{
    struct nlist    *np;
    unsigned        hashval;
    struct nlist    *lookup(const char *);

    if ((np = lookup(name)) == NULL) {  /* not found */
        np = malloc(sizeof(*np));
        if (np == NULL || (np->name = strdup(name)) == NULL)
            return NULL;
        hashval = hash(name);
        np->next = hashtab[hashval];
        hashtab[hashval] = np;
    }
    else                                /* already there */
```

```
        free(np->defn);
    if ((np->defn = strdup(defn)) == NULL)
        return NULL;
    return np;
}
```

A.39 minprintf.c

```
#ifndef lint
static char *sccsid = "@(#)minprintf.c 1.2";
#endif

#include <stdio.h>
#include <stdarg.h>

static void minprintf(const char *, ...);

int
main(void)
{
    minprintf("%s = %f\n", "pi", 3.14159265);
    return 0;
}


/* minprintf:
 * minimal printf with variable argument list
 */
static void
minprintf(const char *fmt, ...)
{
    va_list ap;     /* points to each unnamed arg in turn */
    const char *p, *sval;
    int ival;
    double dval;

    va_start(ap, fmt); /* make ap point to 1st unnamed arg */
    for (p = fmt; *p; ++p) {
        if (*p != '%') {
            (void)putchar(*p);
            continue;
        }

        switch (*++p) {
        case 'd':
            ival = va_arg(ap, int);

            (void)printf("%d", ival);
            break;

        case 'f':
            dval = va_arg(ap, double);

            (void)printf("%f", dval);
            break;
        case 's':
            for (sval = va_arg(ap, char *); *sval; ++sval)
                (void)putchar(*sval);
            break;

        default:
            (void)putchar(*p);
            break;
        }
    }
    va_end(ap);             /* clean up */
}
```

A.40 xcalc1.c

```
#ifndef lint
static char *sccsid = "@(#)xcalc.c 1.2";
#endif

#include <stdio.h>

/* rudimentary calculator */
int
main(void)
{
    double  sum, v;

    sum = 0;
    while (scanf("%lf", &v) == 1)
        (void)printf("\t%.2f\n", sum += v);

    return 0;
}
```

A.41 mycat.c

```
#ifndef lint
static char *sccsid = "@(#)mycat.c 1.2";
#endif

#include <stdio.h>

static void filecopy(FILE *, FILE *);

/* mycat: concatenate files, version 1 */
int
main(int argc, char *argv[])
{
    FILE        *fp;

    if (argc == 1)      /* no args; copy standard input */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                (void)printf("mycat: can't open %s\n", *argv);
                return 1;
            }
            else {
                filecopy(fp, stdout);
                (void)fclose(fp);
            }
    return 0;
}

/* filecopy: copy file ifp to file ofp */
static void
filecopy(FILE *ifp, FILE *ofp)
{
    int     c;

    while ((c = getc(ifp)) != EOF)
        (void)putc(c, ofp);
}
```

A.42 mycat2.c

```
#ifndef lint
static char *sccsid = "@(#)mycat2.c 1.2";
#endif

#include <stdio.h>
#include <stdlib.h>

static void filecopy(FILE *, FILE *);

/* mycat: concatenate files, version 2 */
int
main(int argc, char *argv[])
{
    FILE        *fp;
    char        *prog = argv[0];
                             /* program name for errors */

    if (argc == 1)      /* no args; copy standard input */
        filecopy(stdin, stdout);
    else
        while (--argc > 0)
            if ((fp = fopen(*++argv, "r")) == NULL) {
                (void)fprintf(stderr, "%s: can't open %s\n",
                    prog, *argv);
                exit(1);
            }
            else {
                filecopy(fp, stdout);
                (void)fclose(fp);
            }
    if (ferror(stdout)) {
        (void)fprintf(stderr, "%s: error writing stdout\n",
            prog);
        exit(2);
    }
    return 0;
}

/* filecopy: copy file ifp to file ofp */
static void
filecopy(FILE *ifp, FILE *ofp)
{
    int     c;

    while ((c = getc(ifp)) != EOF)
        (void)putc(c, ofp);
}
```

A.43 mycp.c

```
#ifndef lint
static char *sccsid = "@(#)mycp.c 1.1";
#endif

#include <stdio.h>
#include <unistd.h>

/* copy: copy stdandard input to standard output */
int
main(void)
{
    char    buf[BUFSIZ];
    int     n;

    while ((n = read(STDIN_FILENO, buf, sizeof(buf))) > 0 )
        (void)write(STDOUT_FILENO, buf, n);

    return 0;
}
```

A.44 mygetchar.c

```
#ifndef lint
static char *sccsid = "@(#)mygetchar.c 1.2";
#endif

#include <stdio.h>
#include <unistd.h>

static int mygetchar(void);

int
main(void)
{
    int     c;

    while ((c = mygetchar()) != EOF)
        (void)putchar(c);

    return 0;
}

/* mygetchar: unbuffered single character input */
static int
mygetchar(void)
{
    char    c;

    return (read(STDIN_FILENO, &c, 1) == 1) ?
        (unsigned char)c : EOF;
}
```

A.45 mygetchar2.c

```
#ifndef lint
static char *sccsid = "@(#)mygetchar2.c 1.1";
#endif

#include <stdio.h>
#include <unistd.h>

static int mygetchar(void);

int
main(void)
{
    int     c;

    while ((c = mygetchar()) != EOF)
        (void)putchar(c);

    return 0;
}

/* mygetchar: simple buffered version */
static int
mygetchar(void)
{
    static char buf[BUFSIZ];
    static char *bufp = buf;
    static int  n = 0;

    if (n == 0) {       /* buffer is empty */
        n = read(STDIN_FILENO, buf, sizeof(buf));
        bufp = buf;
    }

    return (--n >= 0) ? (unsigned char)*bufp++ : EOF;
}
```

A.46 mycp1.c

```
#ifndef lint
static char *sccsid = "@(#)mycp1.c 1.3";
#endif

#include <stdio.h>
#include <fcntl.h>
#include <unistd.h>

#define PERMS   0640    /* RW for owner,
                         * R for group,
                         * nothing for other
                         */

static void
error(const char *, ...);

/* mycp: copy f1 to f2 */
int
main(int argc, char *argv[])
{
    int     f1, f2, n;
    char    buf[BUFSIZ];

    if (argc != 3)
        error("usage: %s from to", argv[0]);
    if ((f1 = open(argv[1], O_RDONLY)) == -1)
        error("%s: can't open %s", argv[0], argv[1]);
    if ((f2 = open(argv[2],
            O_WRONLY | O_CREAT | O_TRUNC, PERMS)) == -1)
        error("%s: can't create %s, mode %03o",
            argv[0], argv[2], PERMS);
    while ((n = read(f1, buf, sizeof(buf))) > 0)
        if (write(f2, buf, n) != n)
            error("%s: write error on file %s",
                argv[0], argv[2]);
    return 0;
}

#include <stdarg.h>
#include <stdlib.h>

/* error: print an error message and die */
static void
error(const char *fmt, ...)
{
    va_list args;

    va_start(args, fmt);
    (void)fprintf(stderr, "error: ");
    (void)vfprintf(stderr, fmt, args);
    (void)fprintf(stderr, "\n");
    va_end(args);
    exit(1);
}
```

A.47 mystdio.h

```
/* @(#)mystdio.h 1.2 */

#ifndef MYSTDIO_H
#define MYSTDIO_H

#define MYNULL      0
#define MYEOF       (-1)
#define MYBUFSIZ    1024
#define MYOPEN_MAX  20

typedef struct myiobuf {
    int     cnt;        /* characters left */
    char    *ptr;       /* next character position */
    char    *base;      /* location of buffer */
    int     flag;       /* mode of file access */
    int     fd;         /* file descriptor */
} MYFILE;
extern MYFILE   myiob[MYOPEN_MAX];

#define mystdin     (&myiob[0])
#define mystdout    (&myiob[1])
#define mystderr    (&myiob[2])

enum myflags {
    MY_READ     = 001,      /* file open for reading */
    MY_WRITE    = 002,      /* file open for writing */
    MY_UNBUF    = 004,      /* file is unbuffered */
    MY_EOF      = 010,      /* EOF has occured on this file */
    MY_ERR      = 020       /* error occured on this file */
};

int myfillbuf(MYFILE *);
int myflushbuf(int, MYFILE *);

#define myfeof(p)   (((p)->flag & MY_EOF) != 0)
#define myferror(p) (((p)->flag & MY_ERR) != 0)
#define myfileno(p) ((p)->fd)

#define mygetc(p)   (--(p)->cnt >= 0 \
                    ? (unsigned char)*(p)->ptr++ : myfillbuf(p))
#define myputc(x,p) (--(p)->cnt >= 0 \
                    ? *(p)->ptr++ = (x) : myflushbuf((x),p))

#define mygetchar()     mygetc(mystdin)
#define myputchar(x)    myputc((x), mystdout)

MYFILE *myfopen(const char *, const char *);
```

```
#endif /* MYSTDIO_H */
```

A.48 myfopen.c

```
#ifndef lint
static char *sccsid = "@(#)myfopen.c 1.3";
#endif

#include <fcntl.h>
#include "mystdio.h"

#define PERMS   0640    /* RW for owner,
                         * R for group,
                         * nothing for other
                         */

/* myfopen: open file, return file ptr */
MYFILE *
myfopen(const char *name, const char *mode)
{
    int     fd;
    MYFILE  *fp;

    if (*mode != 'r' && *mode != 'w' && *mode != 'a')
        return MYNULL;
    for (fp = myiob; fp < &myiob[MYOPEN_MAX]; ++fp)
        if ((fp->flag & (MY_READ | MY_WRITE)) == 0)
            break;                  /* found free slot */
    if (fp >= &myiob[MYOPEN_MAX])  /* no free slots */
        return MYNULL;

    if (*mode == 'w')
        fd = open(name, O_RDWR | O_CREAT | O_TRUNC, PERMS);
    else if (*mode == 'a')
        fd = open(name, O_RDWR | O_CREAT | O_APPEND, PERMS);
    else
        fd = open(name, O_RDONLY);
    if (fd == -1)                   /* couldn't access "name" */
        return MYNULL;

    fp->fd = fd;
    fp->cnt = 0;
    fp->base = MYNULL;
    fp->flag = (*mode == 'r') ? MY_READ : MY_WRITE;

    return fp;
}
```

A.49 myfillbuf.c

```
#ifndef lint
static char *sccsid = "@(#)myfillbuf.c 1.1";
#endif

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include "mystdio.h"

/* myfillbuf: open file, return file ptr */
int
myfillbuf(MYFILE *fp)
{
    int     bufsize;

    if ((fp->flag & (MY_READ | MY_EOF | MY_ERR)) != MY_READ)
        return EOF;
    bufsize = (fp->flag & MY_UNBUF) ? 1 : MYBUFSIZ;
    if (fp->base == MYNULL)             /* no buffer yet */
        if ((fp->base = malloc(bufsize)) == MYNULL)
            return EOF;                 /* can't get buffer */
    fp->ptr = fp->base;
    fp->cnt = read(fp->fd, fp->ptr, bufsize);
    if (--fp->cnt < 0) {
        if (fp->cnt == -1)
            fp->flag |= MY_EOF;
        else
            fp->flag |= MY_ERR;
        fp->cnt = 0;
        return EOF;
    }

    return (unsigned char)*fp->ptr++;
}
```

A.50 myiob.c

```
#ifndef lint
static char *sccsid = "@(#)myiob.c 1.2";
#endif

#include "mystdio.h"

MYFILE myiob[MYOPEN_MAX] = {
    { 0, (char *)0, (char *)0, MY_READ,             0},
    { 0, (char *)0, (char *)0, MY_WRITE,            1},
    { 0, (char *)0, (char *)0, MY_WRITE | MY_UNBUF, 2},
};
```

A.51 xmymalloc.c

```
#ifndef lint
static char *sccsid = "@(#)xmymalloc.c 1.2";
#endif

#include <stdio.h>
#include <unistd.h>


typedef long Align;    /* for alignement to long boundary */

union header {              /* block header: */
    struct {
        union header *ptr;  /* next block if on free list */
        unsigned size;      /* size of this block */
    }       s;
    Align   x;              /* force alignment of blocks */
};

typedef union header    Header;

static void *mymalloc(size_t);
static void myfree(void *ap);
static Header *morecore(size_t);


int main(void)
{
    void        *p;

    while ((p = mymalloc(0x100000 - 8)) != NULL)
        (void)printf("%p\n", p);
    return 0;
}


static Header   base;           /* empty list to get started */
static Header   *freep = NULL;  /* start of free list */


/* mymalloc: general-purpose storage allocator */
static void *mymalloc(size_t nbytes)
{
    Header      *p, *prevp;
    unsigned    nunits;

    nunits = (nbytes + sizeof(Header) - 1) /
        sizeof(Header) + 1;
    if ((prevp = freep) == NULL) {      /* no free list yet */
```

```
        base.s.ptr = freep = prevp = &base;
        base.s.size = 0;
    }
    for (p = prevp->s.ptr; ; prevp = p, p = p->s.ptr) {
        if (p->s.size >= nunits) {            /* big enough */
            if (p->s.size == nunits)            /* exactly */
                prevp->s.ptr = p->s.ptr;
            else {                    /* allocate tail end */
                p->s.size -= nunits;
                p += p->s.size;
                p->s.size = nunits;
            }
            freep = prevp;
            return (void *)(p + 1);
        }
        if (p == freep)         /* wrapped around free list */
            if ((p = morecore(nunits)) == NULL)
                return NULL;    /* none left */
    }
}


/* myfree: put block ap in free list */
static void myfree(void *ap)
{
    Header  *bp, *p;

    bp = (Header *)ap - 1;          /* point to block header */
    for (p = freep; bp <= p || bp >= p->s.ptr; p = p->s.ptr)
        if (p >= p->s.ptr && (bp > p || bp < p->s.ptr))
            break;
    /* freed block at start or end of arena */
    if (bp + bp->s.size == p->s.ptr) {
        /* join to upper nbr */
        bp->s.size += p->s.ptr->s.size;
        bp->s.ptr = p->s.ptr->s.ptr;
    }
    else
        bp->s.ptr = p->s.ptr;
    if (p + p->s.size == bp) {
        /* join to lower nbr */
        p->s.size += bp->s.size;
        p->s.ptr = bp->s.ptr;
    }
    else
        p->s.ptr = bp;
    freep = p;
}
```

```
#define NALLOC  1024        /* minimum # of units to request */

static Header *morecore(size_t nu)
{
    Header  *up;

    if (nu < NALLOC)
        nu = NALLOC;

    if ((up = sbrk(nu * sizeof(Header))) == (Header *)(-1))
        return NULL;        /* no space at all */
    up->s.size = nu;
    myfree(up + 1);
    return freep;
}
```

Dodatek B

Referenčni priročnik

B.1 Uvod

Ta priročnik opisuje jezik `C` kot je določen v *ANSI/ISO Standard for the C Programming Language*. Strogo vzeto je to interpretacija standarda, ne standard sam, čeprav smo skrbno pazili, da je ta priročnik zanesljivo vodilo.

Priročnik v glavnem sledi razporeditvi standarda, ki sledi drugi izdaji knjige *The C Programming Language*, ki spet sledi prvi izdaji te knjige, čeprav se organizacija v podrobnostih loči. Z izjemo preimenovanja nekaterih produkcij in dejstva, da nismo formalizirali leksičnih atomov ali predprocesorja, je slovnica, ki je tukaj opisana za jezik v ožjem pomenu besede, enakovredna standardu.

B.2 Leksični dogovori

Program je sestavljen iz ene ali več prevajalnih enot, shranjenih na datotekah. Prevajanje poteka v več fazah, kot je opisano v §B.12. Začetne faze opravljajo leksične transformacije na nizkem nivoju, izvajajo direktive, ki se začenjajo z znakom `#` in opravljajo makro definicije in nadomeščanje. Ko je predprocesiranje §B.12 končano, je program reduciran na zaporedje leksičnih atomov.

B.2.1 Leksični atomi

Obstaja šest razredov leksičnih atomov: ključne besede, imena, konstante, nizi, operatorji in druga ločila. Presledke, horizontalne in vertikalne predelčnike, nove vrste, nove strani ter komentarje, kot jih bomo spodaj opisali (s skupnim imenom beli znaki), prevajalniki ignorirajo razen v kolikor ne služijo za to, da ločijo zaporedne leksične atome med seboj. Bele znake potrebujemo, da ločimo sicer staknjena imena, ključne besede in konstante.

Če je vhodno zaporedje že razbito na leksične atome vse do nekega znaka, je naslednji leksični atom najdaljše zaporedje znakov, ki lahko tvori nek leksičen atom.

B.2.2 Komentarji

Znaka `/*` vpeljeta komentar, ki je zaključen z znakoma `*/`. Komentarji niso gnezdeni in jih prevajalniki ne prepoznavajo znotraj nizov ali znakovnih konstant.

B.2.3 Imena

Ime je zaporedje črk in števk. Prvi znak mora biti črka, podčrtaj _ velja za črko. Velike in male črke so različni znaki. Imena so lahko poljubno dolga in pri internih imenih je vsaj prvih `31` znakov pomembnih; nekatere izvedbe lahko prepoznavajo še več znakov. Interna imena vključujejo imena predprocesorjevih makrojev ter vsa druga imena, ki nimajo eksterne vezave (§B.11.2). Imena z eksterno vezavo so bolj omejena: posamezne izvedbe lahko prepoznavajo samo vodilnih `6` znakov in lahko ignorirajo razliko med velikimi in malimi črkami.

B.2.4 Ključne besede

Naslednja besede so ključne besede in jih ne moremo uporabljati drugače:

```
auto      double   int      struct
break     else     long     switch
case      enum     register typedef
char      extern   return   union
const     float    short    unsigned
continue  for      signed   void
default   goto     sizeof   volatile
do        if       static   while
```

Nekatere izvedbe rezervirajo še dodatna imena, a take izvedbe so s tem postale nestandardne.

B.2.5 Konstante

Obstaja več vrst konstant. Vsaka je svojega tipa; §B.4.2 opisuje osnovne tipe.

konstanta:
  realna-konstanta
  celoštevilčna-konstanta
  enumeracijska-konstanta
  znakovna-konstanta

B.2.5.1 Realne konstante

Realno konstanto tvori celi del, decimalna pika, ulomljeni del, `e` ali `E`, neobvezno predznačen celoštevični eksponent in neobvezna pripona `f`, `F`, `l` ali `L`, ki določa tip. Celi in ulomljeni del sta zaporedji števk. Celi ali ulomljeni del (a ne obeh) lahko opustimo; decimalno piko ali `e` in eksponent (a ne obeh) lahko opustimo. Tip je določen s pripono; `F` ali `f` pomeni tip `float`; `L` ali `l` tip `long double`; sicer je tip `double`.

B.2.5.2 Celoštevilčne konstante

Celoštevilčno konstanto razume prevajalnik kot osmiško, če se začenja s števko nič, sicer pa kot desetiško. Osmiške konstante ne vsebujejo števk `8` ali `9`. Če pred zaporedjem števk stoji `0x` ali `0X` (števka nič), to pomeni šestnajstiško število. Šestnajstiške števke vključujejo `a` ali `A` do `f` ali `F` z vrednostmi `10` do `15`.

Celoštevilski konstanti lahko pripnemo `u` ali `U`, da nakažemo, da je konstanta nepredznačena. Pripnemo lahko tudi črko `l` ali `L`, da nakažemo, da je konstanta `long`.

Tip celoštevilske konstante je odvisen od oblike, vrednosti in pripone. (Glej §B.4 za diskusijo o tipih). Če konstanta nima pripone in je desetiška, je prvega od naslednjih tipov, kamor tako vrednost lahko shranimo: `int`, `long int`, `unsigned long int`. Če konstanta nima pripone in je osmiška ali šestnajstiška, potem je prvega od naslednjih tipov: `int`, `unsigned int`, `long int`, `unsigned long int`. Če ima pripono `u` ali `U`, tedaj `unsigned int`, `unsigned long int`. Če ima pripono `l` ali `L`, tedaj `long int`, `unsigned long int`. Če ima obe priponi, tedaj `unsigned long int`.

B.2.5.3 Enumeracijske konstante

Imena, deklarirana kot enumeratorji (glej §B.8.4), so konstante tipa `int`.

B.2.5.4 Znakovne konstante

Znakovna konstanta je zaporedje enega ali več znakov, vklenjeno med enojne narekovaje, kot `'x'`. Vrednost znakovne konstante z enim samim znakom je numerična vrednost tega znaka v kodni tabeli računalnika, ki program izvaja. Vrednost znakovne konstante z večimi znaki je odvisna od izvedbe.

Znakovne konstante ne vsebujejo znaka `'` ali novih vrst; da bi te znake in še nekatere druge lahko prikazali, uporabljamo naslednja ubežna zaporedja.

nova vrsta	`NL (LF)`	`\n`
horizontalni predelčnik	`HT`	`\t`
vertikalni predelčnik	`VT`	`\v`
povratni znak	`BS`	`\b`
pomik na začetek vrste	`CR`	`\r`
pomik na novo stran	`FF`	`\f`
slišni signal	`BEL`	`\a`
nagibnica	`\`	`\\`
vprašaj	`?`	`\?`
enojni narekovaj	`'`	`\'`
dvojni narekovaj	`"`	`\"`
osmiško število	`ooo`	`\ooo`
šestnajstiško število	`hh`	`\xhh`

Ubežno zaporedje `\ooo` je sestavljeno iz nagibnice, ki ji slede `1`, `2` ali `3` osmiške števke in ki pomenijo vrednost željenega znaka. Običajen primer te konstrukcije je `\0` (, ki mu ne sledi števka), ki pomeni znak `NUL`. Ubežno zaporedje `\xhh` tvorijo nagibnica, črka `x` in zaporedje šestnajstiških števk, ki pomenijo vrednost željenega znaka. Število šestnajstiških števk ni omejeno, a vrednost ustreznega znaka je nedefinirana, če presega vrednost največjega znaka. Za osmiška in šestnajstiška ubežna zaporedja, če izvedba obravnava tip `char` kot predznačen, bo vrednost znaka obravnavana, kot da bi uporabili prisilo (`char`). Če znak, ki sledi nagibnici, ni eden od zgoraj naštetih, je obnašanje nedefinirano.

V nekaterih izvedbah obstaja razširjena množica znakov, ki jih ne moremo prikazati kot tip `char`. Konstanto v tej razširjeni množici pišemo s predpono `L`, na primer `L'x'`, in jo imenujemo široko znakovno konstanto. Taka konstanta je tipa `wchar_t`, to je, celoštevičnega tipa definiranega v standardni glavi `<stddef.h>`. Kot pri običajnih znakovnih konstantah lahko uporabljamo osmiška in šestnajstiška ubežna zaporedja; učinek je nedefiniran, če predpisana vrednost presega maksimalno vrednost, ki se jo da shraniti v tip `wchar_t`.

B.2.6 Nizi

Niz je zaporedje znakov vklenjeno v dvojne narekovaje, kot `"..."`. Niz je tipa večkraten znak in pomnilniškega razreda `static` (glej §B.4 spodaj), ki je inicializiran z danimi znaki. Izvedba se odloči, ali so identični nizi shranjeni ločeno ali ne in obnašanje programa, ki poskuša spremeniti znake v nizu, je nedefinirano.

Nize, ki neposredno slede drug drugemu, prevajalnik stakne v en sam niz. Po končani konkatenaciji prevajalnik doda zlog `\0`, tako da programi, ki pregledujejo niz, lahko najdejo konec niza. V nizu ne moremo napisati nove vrste ali dvojnega narekovaja; če ju hočemo vgraditi v niz, lahko uporabimo ista ubežna zaporedja kot pri znakih.

Kot pri znakovnih konstantah lahko tvorimo široke nize s predpono `L`, kot v `L"..."`. Široki nizi so tipa večkraten `wchar_t`. Konkatenacija običajnega in širokega niza ni definirana.

B.3 Sintaktične oznake

V sintaktičnih pravilih, kot jih uporablja ta priročnik, so sintaktične kategorije zapisane ležeče, dobesedne besede in znaki v Courier fontu. Alternativne kategorije so praviloma navedene vsaka v svoji vrsti; dolgo zaporedje kratkih alternativ je izjemoma našteto v eni vrsti s predhodno frazo "eden od". Neobvezen terminalen ali neterminalen simbol je označen z indeksom $_{opt}$, tako da, na primer,

{ *izraz*$_{opt}$ }

pomeni neobvezen izraz v zavitih oklepajih. Sintaksa je povzeta v §B.13.

B.4 Pomen imen

Imena uporabljamo za vrsto reči: funkcije; značke struktur, unij in enumeracij; komponente struktur ali unij; enumeracijske konstante; `typedef` imena; in objekte. Objekt, ki mu včasih pravimo tudi spremenljivka, je mesto v pomnilniku in njegova interpretacija je predvsem odvisna od dveh atributov: njegovega pomnilniškega razreda in tipa. Pomnilniški razred določa življensko dobo objekta, ki je povezan z identificiranim objektom; tip določa pomen vrednosti, ki jih najdemo v identificiranem objektu. Ime ima tudi doseg, to je območje v programskem tekstu, kjer je ime vidno, in povezovanje, ki določa, kdaj isto ime v drugem dosegu pomeni isti objekt ali funkcijo. Doseg in povezovanje obravnavamo v §B.11.

B.4.1 Pomnilniški razred

Obstajata dva pomnilniška razreda: avtomatičen in statičen. Več ključnih besed, skupaj s kontekstom objektove deklaracije, določa objektov pomnilniški razred. Avtomatični objekti so lokalni bloku (§B.9.3) in izginejo ob izhodu iz bloka. Deklaracije v notranjosti bloka generirajo avtomatične objekte, če pomnilniški razred ni predpisan ali pa če uporabimo določilo `auto`. Objekti, deklarirani kot `register`, so avtomatični in so, če je mogoče, shranjeni v hitrih registrih računalnika.

Statični objekti so lahko lokalni bloku ali pa so eksterni vsem blokom, a vsakem primeru zadrže svoje vrednosti od izhoda do ponovnega vhoda v blok ali funkcijo. Znotraj bloka, vključno z blokom, ki predstavlja jedro funkcije, so statični objekti deklarirani s ključno besedo `static`. Objekti, definirani zunaj vseh blokov, na istem nivoju kot definicije funkcij, so vedno statični. Taki objekti postanejo lokalni prevajalni enoti z uporabo ključne besede `static`; to jim da interno povezovanje. Ti objekti postanejo globalni za ves program s tem, da opustimo ekspliciten pomnilniški razred, ali pa da uporabimo ključno besedo `extern`; to jim da eksterno povezovanje.

B.4.2 Osnovni tipi

Obstaja nekaj osnovnih tipov. Standardna glava `<limits.h>`, opisana v dodatku C, definira največje in najmanjše vrednosti za vsak tip v lokalni izvedbi. Števila, zapisana v dodatku C, pomenijo najmanjše sprejemljive vrednosti.

Objekti, deklarirani kot znaki (`char`), so dovolj široki, da hranijo vsak znak iz nabora znakov v času izvajanja. Če je ta znak resničen znak, je njegova koda nenegativno celo število. V spremenljivke tipa `char` lahko shranimo tudi drugačne vrednosti, a razpon vrednosti, ki jih lahko shranimo, in odgovor na vprašanje, ali so predznačene ali ne, je odvisen od izvedbe.

Nepredznačeni znaki, deklarirani kot `unsigned char`, potrošijo enako količino pomnilnika kot običajni `char`, a so vedno videti nenegativni; eksplicitno predznačeni znaki, deklarirani kot `signed char`, ravno tako potrošijo enako količino pomnilnika kot običajni `char`.

Poleg tipov `char` imamo na razpolago še do tri širine celih števil, deklariranih kot `short int`, `int` in `long int`. Objekti tipa `int` imajo naravno širino, ki jo sugerira arhitektura računalnika; ostale širine so namenjene za posebne potrebe. Širša cela števila pomenijo vsaj toliko prostora kot ožja, a izvedba se lahko odloči, da je običajen `int` isto kot `short int` ali pa `long int`. Celoštevilčni tipi vsi pomenijo predznačena števila, če ni rečeno drugače.

Nepredznačena cela števila, ki jih deklariramo s ključno besedo `unsigned`, ubogajo zakone modularne aritmetike, kjer je modul 2^n, `n` pa je ševilo bitov v reprezentaciji, in tako nepredznačena števila nikoli ne morejo prekoračiti obsega. Množica nenegativnih vrednosti, ki jih lahko shranimo v predznačen objekt, je podmnožica vrednosti, ki jih lahko shranimo v ustrezen nepredznačen objekt in predstavitev skupnih elementov je identična.

Tipi realnih števil `float`, `double` in `long double` so lahko sinonimni, a tisti, ki so kasneje na spisku, so vsaj tako natančni kot tisti na začetku seznama.

Enumeracije so posebni tipi, ki imajo celoštevilčne vrednosti; vsaki enumeraciji pripada množica konstant z imeni (§B.8.4). Enumeracije se obnašajo kot cela števila, a prevajalniki običajno protestirajo s svarilom, če objektu nekega enumeracijskega tipa priredimo kaj drugega razen njegovih konstant ali izraza njegovega tipa.

Ker objekte teh tipov lahko interpretiramo kot števila, jih bomo imenovali aritmetični tipi. Tipe `char`, `int` vseh širin, vsak predznačen ali ne, in tudi enumeracijske tipe, bomo s skupnim imenom klicali celi tipi. Tipe `float`, `double` in `long double` bomo imenovali realni tipi.

Tip `void` pomeni prazno množico vrednosti. Uporabljamo ga kot tip, ki ga vrnejo funkcije, ki ne vrnejo nobene vrednosti.

B.4.3 Izpeljani tipi

Poleg osnovnih tipov imamo potencialno neskončen razred izpeljanih tipov, ki jih izpeljemo iz osnovnih tipov z naslednjimi mehanizmi:

- večkratni objekti danega tipa;
- funkcije, ki vračajo objekte danega tipa;
- kazalci na objekte danega tipa;
- strukture, ki vsebujejo zaporedja objektov različnih tipov;
- unije, ki so sposobne hraniti enega od objektov raznih tipov.

Te mehanizme lahko v splošnem uporabljamo rekurzivno.

B.4.4 Kvalifikacije tipov

Tip objekta ima lahko še dodatne kvalifikacije. Če deklariramo, da je objekt konstanten, `const`, napovemo, da se njegova vrednost ne bo spreminjala; če deklariramo objekt `volatile`, napovemo, da ima posebne lastnosti, pomembne za optimizacijo. Nobena kvalifikacija ne vpliva na množico vrednosti ali na aritmetične lastnosti objekta. Kvalifikacije obravnavamo v §B.8.2.

B.5 Objekti in *lvrednosti*

Objekt je področje pomnilnika z imenom; *lvrednost* je izraz, ki se nanaša na objekt. Očiten primer *lvrednosti* je ime objekta primernega tipa in pomnilniškega razreda. Nekateri operatorji vračajo *lvrednosti*: na primer, če je `E` izraz kazalčnega tipa, je `*E` *lvrednost*, ki se nanaša na objekt, na katerega `E` kaže. Izraz *lvrednost* prihaja iz prirejanja `E1 = E2`, kjer mora biti levi operand `E1` *lvrednost*. Pri vsakem operatorju bomo navedli, če pričakuje operande, ki so *lvrednosti*, in če vrne *lvrednost*.

B.6 Pretvarjanja

Nekateri operatorji lahko, odvisno od njihovih operandov, povzroče pretvarjanje vrednosti operanda iz enega tipa v drugega. Ta razdelek razlaga rezultate, ki jih lahko pričakujemo od takih pretvarjanj. §B.6.5 povzema pretvarjanja, ki jih terja večina običajnih operatorjev; ta pretvarjanja bomo dopolnili, kot bo potrebno, pri obravnavi vsakega operatorja posebej.

B.6.1 Celoštevična promocija

Količino `char`, `short int` ali bitno polje, vsak od njih predznačen ali ne, lahko uporabimo v izrazu, kjer smemo uporabiti `int`. Če v `int` lahko shranimo vse vrednosti originalnega tipa, tedaj bo prevajalnik pretvoril originalen tip v `int`; če ne, ga bo pretvoril v `unsigned int`. Temu procesu pravimo celoštevilčna promocija.

B.6.2 Celoštevilčna pretvarjanja

Katerokoli celo število pretvori program v dan nepredznačen tip tako, da najde najmanjše nenegativno število, ki je kongruentno danemu številu, modulo ena več kot največje število, ki ga lahko shranimo v nepredznačen tip. V dvojiškem komplementu to pomeni rezanje odvečnih bitov na levi, če je ciljni tip ožji, dodajanje ničel na levi v primeru da je nepredznačeni ciljni tip širši in dodajanje kopij prvega bita na levi v primeru da je predznačeni ciljni tip širši.

Kadarkoli neko število pretvorimo v nek predznačen tip, ostane vrednost nespremenjena, če je to mogoče, sicer pa je obnašanje odvisno od izvedbe.

B.6.3 Cela in realna števila

Kadarkoli mora računalnik pretvoriti realno število v celo, bo odrezal neceli del; če dobljenega celega števila ni mogoče shraniti v celi tip, je obnašanje nedefinirano. Specialno, rezultat pretvarjanja negativnega realnega števila v nepredznačeno celo število je nedefiniran.

Kadar pretvarjamo celo število v realno in je vrednost v dosegu realnih števil, a celega števila ne moremo natančno predstaviti kot realno število, dobimo prvo večjo ali prvo manjšo predstavljivo celo vrednost. Če je realen rezultat izven dosega realnih števil, je obnašanje nedefinirano.

B.6.4 Realna števila

Kadar manj natančno realno število pretvorimo v enako ali bolj natančno realno število, ostane vrednost nespremenjena. Kadar bolj natančno vrednost pretvarjamo v manj natančno in je rezultat predstavljiv, dobimo naslednje večje ali naslednje manjše predstavljivo število. Če je rezultat zunaj dosega, je obnašanje nedefinirano.

B.6.5 Aritmetična pretvarjanja

Mnogo operatorjev povzroča pretvarjanja in nam da rezultat na podoben način. Učinek je ta, da prevedemo operanda na skupen tip, ki je potem tudi tip rezultata. Ta proces se imenuje *običajno aritmetično pretvarjanje.*

Najprej, če je en operand `long double`, se drugi prevede v `long double`.
Sicer, če je en operand `double`, se drugi prevede v `double`.
Sicer, če je en operand `float`, se drugi prevede v `float`.
Sicer opravimo celoštevilčno promocijo na opeh operandih; če je potem en operand `unsigned long int`, se drugi pretvori v `unsigned long int`.
Sicer, če je en operand `long int` in drugi `unsigned int`, je postopek odvisen od tega, ali lahko `long int` hrani vse vrednosti `unsigned int`; če lahko, se `unsigned int` pretvori v `long int`; če ne, se oba pretvorita v `unsigned long int`.
Sicer, če je en operand `long int`, se drugi pretvori v `long int`.
Sicer, če je en operand `unsigned int`, se drugi pretvori v `unsigned int`.
Sicer sta oba operanda tipa `int`.

B.6.6 Kazalci in cela števila

Celoštevilčen izraz lahko prištejemo ali odštejemo od kazalca; v tem primeru se celoštevilčen izraz pretvori, kot je opisano v diskusiji operatorja seštevanja (§B.7.7).

Dva kazalca na dva objekta istega tipa, v isti večkratni vrednosti, lahko odštejemo. Rezultat je celo število kot je opisano ob diskusiji operatorja odštevanja (§B.7.7).

Konstanten celoštevilčen izraz z vrednostjo nič, ali tak izraz prisiljen v tip `void *`, lahko pretvorimo s prisilo, s prirejanjem ali s primerjanjem v kazalec poljubnega tipa. Rezultat je prazen kazalec, ki je enak vsakemu drugemu praznemu kazalcu na isti tip, a različen od vsakega kazalca na funkcijo ali objekt.

Tudi nekatera druga pretvarjanja, ki zajemajo kazalce, so dovoljena, a imajo od izvedbe odvisne aspekte. Dosežemo jih z eksplicitnim operatorjem za pretvorbo tipa ali s prisilo (§§B.7.5 in B.8.8).

Kazalec lahko pretvorimo v celo število, dovolj široko, da ga lahko hrani; potrebna širina je odvisna od izvedbe. Tudi funkcija preslikavanja je odvisna od izvedbe.

Kazalec na en tip lahko pretvorimo v kazalec na nek drug tip. Dobljeni kazalec lahko povzroči naslovno izjemo, če dobljeni kazalec ne kaže na pravilno poravnan objekt. Jezik garantira, da lahko kazalec na zahtevno poravnan objekt pretvorimo v kazalec na manj zahtevno poravnan objekt in nazaj brez sprememb. Pojem poravnavanja je odvisen od izvedbe, a objekti tipa `char` imajo najmanj zahtevno poravnavanje. Kot je opisano v §B.6.8, kazalec lahko pretvorimo v tip `void *` in nazaj brez sprememb.

Končno, kazalec na funkcijo lahko pretvorimo v kazalec na drugačno funkcijo. Klic funkcije, predpisane s pretvorjenim kazalcem, je odvisen od izvedbe; če pa pretvorjen kazalec pretvorimo nazaj v originalen tip, je rezultat identičen originalnemu kazalcu.

B.6.7 Void

(Neobstoječe) vrednosti `void` objekta ne moremo uporabiti v nobenem kontekstu, niti ne moremo `void` vrednosti prisiliti v kakšen ne `void` tip. Ker izraz tipa `void` pomeni neobstoječo vrednost, ga lahko uporabimo samo tam, kjer vrednosti ne potrebujemo, na primer v izraznem stavku (§B.9.2) ali kot levi operand vejičnega operatorja (§B.7.18).

Izraz lahko pretvorimo v tip `void` s prisilo. Na primer, prisila `(void)` dokumentira, da v izraznem stavku vrednosti funkcije ne potrebujemo.

B.6.8 Kazalci na void

Vsak kazalec lahko pretvorimo v tip `void *` brez izgube informacije. Če tak `void *` pretvorimo nazaj v originalen kazalec, dobimo originalno vrednost. Za razliko od pretvorb enih kazalcev v druge, ki smo jih obravnavali v §B.6.6 in ki terjajo uporabo prisile, kazalce na `void` lahko prosto prirejamo in primerjamo.

B.7 Izrazi

Prioriteta izraznih operatorjev je ista kot zaporedje glavnih podrazdelkov v tem razdelku, tisti z najvišjo prioriteto najprej. Tako so na primer operandi operatorja `+` tisti izrazi, ki so definirani v §§§B.7.1-§B.7.6. Znotraj vsakega podrazdelka imajo operatorji isto prioriteto. Pri vsakem operatorju bomo podali asociativnost z leve ali z desne. Slovnica vključuje prioriteto in asociativnost operatorjev; povzeta je v §B.13.

Prioriteta in asociativnost so v polnosti predpisani, a vrstni red računanja, z nekaterimi izjemami, ni predpisan, celo če podizrazi vključujejo stranske učinke. To se pravi, razen če definicija operatorja ne garantira računanja v nekem vrstem redu, lahko vsaka izvedba organizira računanje po svoje. Seveda pa vsak operator kombinira svoje operande v skladu z razpoznavanjem izraza, kjer se operator nahaja.

Obravnavanje prekoračitve obsega, deljenja z nič in drugih izjem pri računanju vrednosti izrazov z jezikom ni definirano. Večina obstoječih izvedb `C` ignorira prekoračitev obsega pri računanju s predznačenimi celimi izrazi in pri prirejanju, a to obnašanje ni uzakonjeno. Obravnava deljenja z nič in vseh izjem pri realnih številih se od izvedbe do izvedbe spreminja; včasih se to obnašanje da nastaviti z nestandardno knjižnično funkcijo.

B.7.1 Generiranje kazalcev

Če je tip izraza ali podizraza "večkraten T", za nek tip T, tedaj je vrednost takega izraza kazalec na prvi objekt v večkratni vrednosti in tip izraza se spremeni v "kazalec na T". Do te spremembe ne pride, če je izraz operand unarnega operatorja `&`, operatorja `++`, `--`, `sizeof`, ali pa levi operand operatorja prirejanja ali operatorja pika. Podobno, izraz tipa "funkcija, ki vrne T", razen pri `&` operatorju, postane "kazalec na funkcijo, ki vrne T". Izraz, ki je pretrpel eno od teh sprememb, ni več *lvrednost.*

B.7.2 Primarni izrazi

Primarni izrazi so imena, konstante, nizi ali izrazi v oklepajih.

primarni-izraz:
  ime
  konstanta
  niz
  (*izraz*)

Ime je primaren izraz, če je to ime primerno definirano, kot je spodaj opisano. Njegov tip je določen z definicijo. Ime je lvrednsot, če se nanaša na objekt (B.5) in njegov tip je aritmetičen, struktura, unija ali kazalec.

Konstanta je primaren izraz. Njen tip je odvisen od oblike kot je opisano v §B.2.5.

Niz je primaren izraz. Njegov tip je originalno "večkraten `char`" (za široke nize "večkraten `wchar_t`"), a po pravilu §B.7.1 je običajno spremenjen v "kazalec na `char`" (kazalec na `wchar_t`) in rezultat je kazalec na prvi znak v nizu. Do te spremembe ne pride pri določenih inicializatorjih; glej §B.8.7.

Izraz v oklepajih je primaren izraz čigar tip in vrednost sta ista kot pri neobjetem izrazu. Prisotnost oklepajev ne spremeni dejstva, da izraz je ali ni *lvrednost*.

B.7.3 Priponski izrazi

Operatorji v priponskih izrazih asociirajo z leve na desno.

priponski-izraz:
  primarni-izraz
  priponski-izraz `[` *izraz* `]`
  priponski-izraz `(` *seznam-argumentnih-izrazov*$_{opt}$ `)`
  priponski-izraz `.` *ime*
  priponski-izraz `->` *ime*
  priponski-izraz `++`
  priponski-izraz `--`

seznam-argumentnih-izrazov:
  argumentni-izraz
  seznam-argumentnih-izrazov `,` *argumentni-izraz*

B.7.3.1 Reference na večkratno vrednost

Priponski izraz, ki mu sledi izraz v oglatih oklepajih, je priponski izraz, ki naznačuje indeksirano referenco na večkratno vrednost. Eden od obeh izrazov mora biti tipa "kazalec na `T`", kjer je `T` nek tip, drugi pa mora biti celega tipa; tip indeksiranega izraza je `T`. Izraz `E1[E2]` je (po definiciji) identičen z izrazom `*((E1)+(E2))`. Glej §B.8.6.2 za nadaljnjo diskusijo.

B.7.3.2 Klici funkcij

Klic funkcije je priponski izraz, imenovan označevalec funkcije, ki mu sledi par okroglih oklepajev, ki vsebuje, morda prazen, seznam argumentnih izrazov, ločenih z vejico (§B.7.17), in ki pomenijo argumente funkcije. Če priponski izraz tvori ime, za katero v trenutnem dosegu ni deklaracije, je ime implicitno deklarirano, kot da bi bila v najbolj notranjem bloku, ki vsebuje funkcijski klic, napisana deklaracija

```
extern int ime();
```

Priponski izraz (po morebitni implicitni deklaraciji in generiranju kazalcev, §B.7.1) mora biti tipa "kazalec na funkcijo, ki vrača `T`", za nek tip `T`, in vrednost takega klica je tipa `T`.

Termin argument uporabljamo za izraz, ki ga podamo funkciji ob klicu; termin parameter uporabljamo za vhodni objekt (ali njegovo ime), ki ga dobi definicija funkcije ali pa je opisan z deklaracijo funkcije. Termina "formalni argument (parameter)" in "dejanski argument (parameter)" včasih uporabljamo za isto razlikovanje.

Kot del priprave na klic funkcije, prevajalnik napravi kopijo vsakega argumenta; vsi argumenti se prenašajo striktno po vrednosti. Funkcija lahko spremeni vrednosti svojih parametrov, ki so kopije argumentnih izrazov, a te spremembe ne morejo vplivati na vrednosti argumentov. Seveda pa je možno prenesti kazalec v funkcijo in funkcija lahko spremeni vrednost, na katero kazalec kaže.

Obstajata dva stila, kako lahko deklariramo funkcije. V novem stilu so tipi parametrov eksplicitni in so del tipa funkcije; takšna deklaracija se imenuje tudi funkcijski prototip. V starem stilu tipi parametrov niso navedeni. Deklaracije funkcij obravnavamo v §§B.8.6.3 in §B.10.1.

Če je funkcijska deklaracija v dosegu klica v starem stilu, potem uporabimo avtomatično promocijo argumentov: celoštevilčna promocija vsakega celoštevilčnega argumenta in vsak `float` argument se pretvori v `double`. Učinek klica je nedefiniran, če je število argumentov različno od števila parametrov v definiciji funkcije ali če se tipi argumentov po promociji ne ujemajo s tipi ustreznih parametrov. Ujemanje tipov je odvisno od stila. Če je funkcija definirana v starem stilu, tedaj se primerjata promoviran tip argumenta in promoviran tip parametra; če je funkcija definirana v novem stilu, se mora promoviran tip argumenta ujemati s tipom parametra, brez promocije.

Če je deklaracija funkcije v dosegu klica v novem stilu, potem se argumenti pretvorijo, kot pri prirejanju, v tipe ustreznih parametrov v prototipu. Število argumentov mora biti isto kot število eksplicitno opisanih parametrov, razen če se seznam parametrov v deklaraciji ne končuje s tropičjem .... V tem primeru mora biti število argumentov večje ali enako številu parametrov; argumenti na koncu seznama, onkraj eksplicitno tipiziranih parametrov, so podvrženi avtomatični promociji argumentov, kot je opisana v predhodnem paragrafu. Če je definicija funkcije v starem stilu, potem se mora tip vsakega parametra, vidnega ob klicu, ujemati z ustreznim parametrom v definiciji, potem ko je tip parametra v definiciji pretrpel argumentno promocijo.

Vrstni red izračunavanja argumentov ni predpisan; različni prevajalniki ubirajo različne poti. V vsakem primeru pa se vsi argumenti in označevalec funkcije do kraja izračunajo, vključno z vsemi stranskimi učinki, preden funkcija prevzame kontrolo. Vsako funkcijo lahko uporabljamo rekurzivno.

B.7.3.3 Reference na strukture

Priponski izraz, ki mu sledi pika in ime, je priponski izraz. Operand pred piko mora biti struktura ali unija, ime za piko mora biti ime komponente v strukturi ali uniji. Vrednost je imenovana komponenta strukture ali unije, tip je tip komponente. Izraz je *lvrednost*, če je levi operand pike *lvrednost* in če desni operand pike ni večkratna vrednost.

Priponski izraz, ki mu sledita puščica (zgrajena iz `-` in `>`) in ime, je priponski izraz. Operand pred puščico mora biti kazalec na strukturo ali unijo, ime za puščico mora biti ime komponente v strukturi ali uniji. Rezultat se nanaša na imenovano komponento strukture ali unije, tip je tip komponente; rezultat je *lvrednost*, če komponenta ni večkratna vrednost.

Tako je `E1->MOS` isto kot `(*E1).MOS`. Strukture in unije obravnavamo v §B.8.3.

B.7.3.4 Priponsko povečevanje

Priponski izraz, ki mu sledi `++` ali `--` operator, je priponski izraz. Vrednost izraza je vrednost operanda. Potem, ko je ta vrednost pripravljena, se operand poveča (`++`) ali zmanjša (`--`) za `1`. Operand mora biti *lvrednost*; glej diskusijo aditivnih operatorjev (§B.7.7) in prirejanje (§B.7.17) za nadaljnje omejitve operanda in podrobnosti operacije. Rezultat ni *lvrednost*.

B.7.4 Unarni operatorji

Izrazi z unarnimi operatorji asociirajo z desne proti levi.

unarni-izraz:
 priponski-izraz
 `++` *unarni-izraz*
 `--` *unarni-izraz*
 unarni-operator prisiljeni-izraz
 `sizeof` *unarni-izraz*
 `sizeof` `(` *ime-tipa* `)`

unarni-operator: eden od
 `&` `*` `+` `-` `~` `!`

B.7.4.1 Operatorji predponskega povečevanja

Unarni izraz, pred katerim stoji `++` ali `--` operator, je unaren izraz. Operand se poveča (`++`) ali zmanjša (`--`) za `1`. Vrednost izraza je vrednost po povečnaju (zmanjšanju). Operand mora biti *lvrednost*; glej diskusijo aditivnih operatorjev (§B.7.7) in prirejanje (§B.7.17) za nadaljnje omejitve operanda in podrobnosti operacije. Rezultat ni *lvrednost*.

B.7.4.2 Adresni operator

Unarni operator `&` vrne naslov svojega operanda. Operand mora biti *lvrednost*, ki ni niti bitno polje, niti objekt, deklariran kot `register`, ali pa mora biti funkcijskega tipa. Rezultat je kazalec na objekt ali funkcijo, na katero se sklicuje *lvrednost*. Če je tip operanda `T`, je tip rezultata "kazalec na `T`".

B.7.4.3 Operator indirekcije

Unarni operator `*` označuje indirekcijo in vrne objekt ali funkcijo, na katero operand kaže. Rezultat je *lvrednost*, če je izraz kazalec na objekt aritmetičega tipa, strukturo, unijo ali kazalec. Če je tip izraza "kazalec na `T`", tedaj je tip rezultata `T`.

B.7.4.4 Unarni plus operator

Operand unarnega `+` operatorja mora biti aritmetičnega ali kazalčnega tipa in rezultat je vrednost operanda. Celoštevilčen argument pretrpi celoštevilčno promocijo. Tip rezultata je tip promoviranega operanda.

B.7.4.5 Unarni minus operator

Operand unarnega `-` operatorja mora biti aritmetičnega tipa in rezultat je nasprotna vrednost operanda. Celoštevilčen argument pretrpi celoštevilčno promocijo. Nasprotno vrednost nepredznačenega argumenta dobimo z odštevanjem promovirane vrednosti od največje vrednosti promoviranega tipa plus `1`; nasprotna vrednost števila nič je število nič. Tip rezultata je tip promoviranega operanda.

B.7.4.6 Operator eniškega komplementa

Operand operatorja `~` mora biti celoštevilčnega tipa in rezultat je eniški komplement operanda. Argument pretrpi celoštevilčno promocijo. Eniški komplement nepredznačenega argumenta dobimo z odštevanjem promovirane vrednosti od največje vrednosti promoviranega tipa. Če je operand predznačen, dobimo rezultat tako, da pretvorimo promoviran operand v ustrezen nepredznačen tip, uporabimo `~` operator in pretvorimo rezultat nazaj v predznačeni tip. Tip rezultata je tip promoviranega operanda.

B.7.4.7 Operator logične negacije

Operand operatorja `!` mora biti aritmetičnega tipa ali nek kazalec in rezultat je `1`, če je vrednost operanda enaka `0`, sicer pa `0`. Razultat je tipa `int`.

B.7.4.8 Sizeof operator

Operator `sizeof` vrne število zlogov, ki jih potrebujemo, da shranimo objekt tipa njegovega operanda. Operand je bodisi nek izraz, katerega vrednost se ne izračuna, ali pa ime tipa v oklepajih. Če `sizeof` uporabimo na tipu `char`, je rezultat `1`; če ga uporabimo na večkratni vrednosti, je rezultat število zlogov, ki jih zaseda večkratna vrednost. Če `sizeof` uporabimo na strukturi ali uniji, je rezultat število zlogov v objektu, vključno z neimenovanimi luknjami, ki nastanejo vsled poravnavanja na velikost, ki je potrebna za tlakovanje večkratne vrednosti: velikost `n`-kratne vrednosti je `n` krat velikost enega elementa. Operatorja ne moremo uporabiti na funkcijskem tipu, na nepopolnem tipu in na bitnem polju. Rezultat je nepredznačena celoštevilčna konstanta; tip si izbere izvedba. Standardna glava `<stddef.h>` (glej dodatek C) definira ta tip kot `size_t`.

B.7.5 Prisile

Unaren izraz, pred katerim stoji ime tipa v okroglih oklepajih, povzroči pretvorbo vrednosti izraza v imenovani tip.

prisiljeni-izraz:
unarni-izraz
(*ime-tipa*) *prisiljeni-izraz*

Tej konstrukciji pravimo prisila. Imena tipov so opisana v §B.8.8. Učinki pretvarjanja so opisani v §B.6. Prisiljeni izraz ni *lvrednost*.

B.7.6 Multiplikativni operatorji

Multiplikativni operatorji `*`, `/` in `%` asociirajo z leve na desno.

multiplikativni-izraz:
  prisiljeni-izraz
  multiplikativni-izraz `*` *prisiljeni-izraz*
  multiplikativni-izraz `/` *prisiljeni-izraz*
  multiplikativni-izraz `%` *prisiljeni-izraz*

Operanda operatorjev `*` in `/` morata biti aritmetičnega tipa; operanda operatorja `%` morata biti celoštevilčnega tipa. Operanda pretrpita običajne aritmetične pretvorbe in določata tip rezultata.

Binarni operator `*` pomeni množenje.

Binarni operator `/` da kvocient, operator `%` pa ostanek pri deljenju levega operanda z desnim; če je desni operand `0`, je rezultat nedefiniran. Sicer pa je vedno res, da je `(a/b)*b + a%b` enako `a`. Če sta oba operanda nenegativna, potem je ostanek nenegativen in manjši od divizorja; če ne, je zagotovljeno samo, da je absolutna vrednost ostanka manjša od absolutne vrednosti divizorja.

B.7.7 Aditivna operatorja

Aditivna operatorja `+` in `-` asociirata z leve na desno. Če sta operanda aritmentičnega tipa, pretrpita običajne aritmetične pretvorbe. Za vsak operator obstaja še nekaj dodatnih kombinacij tipov.

aditivni-izraz:
  multiplikativni-izraz
  aditivni-izraz `+` *multiplikativni-izraz*
  aditivni-izraz `-` *multiplikativni-izraz*

Rezultat operatorja `+` je vsota obeh operandov. Kazalec na objekt v večkratni vrednosti in vrednost kateregakoli celoštevilčnega tipa lahko seštejemo. Celoštevilčno vrednost prevajalnik najprej pretvori v odmik tako, da jo pomnoži z velikostjo objekta, na katerega kazalec kaže. Vsota je kazalec istega tipa kot originalni kazalec in kaže na nek drug objekt v isti večkratni vrednosti, pravilno odmaknjen od originalnega objekta. Tako, če je `P` kazalec na objekt v večkratni vrednosti, je izraz `P+1` kazalec na naslednji objekt v večkratni vrednosti. Če kazalec, ki ponazarja vsoto, pokaže zunaj meja večkratne vrednosti, z izjemo prvega položaja onkraj konca večkratne vrednosti, je rezultat nedefiniran.

Rezultat operatorja `-` je razlika obeh operandov. Vrednost poljubnega celoštevilčnega tipa lahko odštejemo od kazalca in tedaj veljajo ista pretvarjanja in omejitve kot pri seštevanju.

Če odštejemo dva kazalca na objekte istega tipa, je rezultat predznačena celoštevilčna vrednost, ki ponazarja razmik med objektoma, na katera kazalca kažeta. Kazalca na dva zaporedna objekta se razlikujeta za `1`. Tip rezultata je odvisen od izvedbe, in je definiran kot `ptrdiff_t` v standardni glavi `<stddef.h>`. Vrednost razlike je nedefinirana, če kazalca ne kažeta na dva objekta v isti večkratni vrednosti; vendar, če P kaže na zadnji element večkratne vrednosti, tedaj ima `(P+1)-P` vrednost `1`.

B.7.8 Operatorja pomika

Operatorja pomika `<<` in `>>` asociirata z leve na desno. Za oba operatorja morata biti oba operanda celoštevilčna in sta podvržena celoštevilčni promociji. Tip rezultata je tip promoviranega levega operanda. Rezultat je nedefiniran, če je desni operand negativen ali večji ali enak številu bitov v tipu levega izraza.

pomični-izraz:
  aditivni-izraz
  pomični-izraz `<<` *aditivni-izraz*
  pomični-izraz `>>` *aditivni-izraz*

Vrednost izraza `E1<<E2` je `E1` (interpretirana kot bitni vzorec), pomaknjena v levo za `E2` bitov; če rezultat ne prekorači obsega, je to enakovredno množenju z $2^{\mathtt{E2}}$. Vrednost izraza `E1>>E2` je `E1`, pomaknjena v desno za `E2` bitov. Desni pomik je enakovreden deljenju z $2^{\mathtt{E2}}$, če je `E1` nepredznačen ali pa če ima vsaj nenegativno vrednost; sicer pa je rezultat odvisen od izvedbe.

B.7.9 Relacijski operatorji

Relacijski operatorji asociirajo z leve na desno, a to dejstvo ni koristno; `a<b<c` prepozna razčlenjevalnik kot `(a<b)<c` in `a<b` ima vrednost `0` ali `1`.

relacijski-izraz:
  pomični-izraz
  relacijski-izraz `<` *pomični-izraz*
  relacijski-izraz `>` *pomični-izraz*
  relacijski-izraz `<=` *pomični-izraz*
  relacijski-izraz `>=` *pomični-izraz*

Operatorji `<` (manjši kot), `>` (večji kot), `<=` (manjši ali enak) in `>=` (večji ali enak) vsi vrnejo vrednost `0`, če navedeni pogoj ni izpolnjen in `1`, če je. Rezultat je tipa `int`. Aritmetična operanda pretrpita običajne aritmetične pretvorbe. Kazalce na objekte istega tipa lahko primerjamo; izid je odvisen od relativnega položaja objektov, na katere kazalca kažeta, v adresnem prostoru. Primerjava kazalcev je definirana samo za dele istega objekta: če kazalca kažeta na isti enostaven objekt, tedaj primerjanje kazalcev pravi, da sta kazalca enaka; če kazalca kažeta na komponente iste strukture, tedaj je kazalec na komponento, ki je deklarirana kasneje, večji; če kazalca kažeta na komponente iste unije, tedaj primerjanje kazalcev pravi, da sta kazalca enaka; če kazalca kažeta na dve komponenti iste večkratne vrednosti, tedaj je rezultat primerjanja tak, kot da bi primerjali ustrezna indeksa. Če `P` kaže na zadnji element večkratne vrednosti, tedaj je `P+1` večji od `P`, čeprav `P+1` kaže že zunaj večkratne vrednosti. V ostalih primerih primerjanje kazalcev ni definirano.

B.7.10 Operatorja enakosti

enakostni-izraz:
  relacijski-izraz
  enakostni-izraz `==` *relacijski-izraz*
  enakostni-izraz `!=` *relacijski-izraz*

Operatorja `==` (enak) in `!=` (neenak) sta analogna relacijskim operatorjem, razen da imata nižjo prioriteto. (Tako je `a<b == c<d` enako `1`, če imata `a<b` in `c<d` isto logično vrednost.)

Operatorja enakosti ubogata ista pravila kot relacijski operatorji, a dopuščata dodatne možnosti: kazalec lahko primerjamo s konstatnim celoštevičnim izrazom z vrednostjo `0` ali s kazalcem na `void`. Glej §B.6.6.

B.7.11 Bitni IN operator

IN-izraz:
  enakostni-izraz
  IN-izraz `&` *enakostni-izraz*

Operanda pretrpita običajna aritmetična pretvarjanja; rezultat je IN funkcija po bitih, uporabljena na argumentih. Operator lahko uporabimo le na celoštevilčnih operandih.

B.7.12 Bitni ekskluzivni ALI operator

ekskluzivni-ALI-izraz:
  IN-izraz
  ekskluzivni-ALI-izraz `^` *IN-izraz*

Operanda pretrpita običajna aritmetična pretvarjanja; rezultat je ekskluzivni ALI funkcija po bitih, uporabljena na argumentih. Operator lahko uporabimo le na celoštevilčnih operandih.

B.7.13 Bitni inkluzivni ALI operator

inkluzivni-ALI-izraz:
  ekskluzivni-ALI-izraz
  inkluzivni-ALI-izraz `|` *ekskluzivni-ALI-izraz*

Operanda pretrpita običajna aritmetična pretvarjanja; rezultat je inkluzivni ALI funkcija po bitih, uporabljena na argumentih. Operator lahko uporabimo le na celoštevilčnih operandih.

B.7.14 Logični IN operator

logični-IN-izraz:
  inkluzivni-ALI-izraz
  logični-IN-izraz `&&` *inkluzivni-ALI-izraz*

Operator `&&` asociira z leve na desno. Operator vrne `1`, če imata oba operanda vrednost različno od nič, sicer pa `0`. Za razliko od `&` operator `&&` garantira računanje z leve na desno: levi operand se izračuna, skupaj z vsemi stranskimi učinki; če je njegova vrednost `0`, ima izraz vrednost `0`. V nasprotnem primeru je vrednost izraza vrednost njegovega desnega operanda.

Operanda ni treba, da sta istega tipa, a vsak mora biti aritmetičnega tipa ali kazalec. Rezultat je tipa `int`.

B.7.15 Logični ALI operator

logični-ALI-izraz:
    logični-IN-izraz
    logični-ALI-izraz `||` *logični-IN-izraz*

Operator `||` asociira z leve na desno. Operator vrne `1`, če ima vsaj en operand vrednost različno od nič, sicer pa `0`. Za razliko od `|` operator `||` garantira računanje z leve na desno: levi operand se izračuna, skupaj z vsemi stranskimi učinki; če je njegova vrednost različna od `0`, ima izraz vrednost `1`. V nasprotnem primeru je vrednost izraza vrednost njegovega desnega operanda.

Operanda ni treba, da sta istega tipa, a vsak mora biti aritmetičnega tipa pa kazalec. Rezultat je tipa `int`.

B.7.16 Pogojni operator

pogojni-izraz:
    logični-ALI-izraz
    logični-ALI-izraz ? *izraz* : *pogojni-izraz*

Prvi izraz se izračuna, skupaj z vsemi stranskimi učinki; če je ta vrednost različna od `0`, je rezultat vrednost drugega izraza, sicer pa je rezultat vrednost tretjega izraza. Samo eden od drugega in tretjega izraza se izračuna. Če sta drugi in tretji operand aritmetična, tedaj pretrpita običajno aritmetično pretvorbo, da ju pretvorimo na skupni tip in to je tudi tip rezultata. Če sta oba tipa `void`, strukturi ali uniji istega tipa ali kazalca na objekte istega tipa, je tip rezultata ta skupni tip. Če je eden od obeh operandov kazalec drugi pa konstanta `0`, se `0` pretvori v ta kazalčni tip in to je tudi tip rezultata. Če je en operand kazalec na `void` drugi pa nek drug kazalec, se ta kazalec protvori v kazalec na `void` in to je tip rezultata.

Pri primerjanju tipov kazalcev so vsi kvalifikatorji (§B.8.2) brezpredmetni, a rezultat podeduje kvalifikatorje z obeh strani pogojnega izraza.

B.7.17 Prirejanja

Obstaja vrsta operatorjev prirejanja; vsi asociirajo z desne na levo.

prirejanje:
    pogojni-izraz
    unarni-izraz operator-prirejanja prirejanje

operator prirejanja: eden od
    `=  *=  /=  %=  +=  -=  <<=  >>=  &=  ^=  |=`

Vsi operatorji prirejanja zahtevajo, da je levi operand *lvrednost* in ta *lvrednost* mora biti modifikabilna: ne sme biti večkratna vrednost, nepopoln tip ali funkcija. Njen tip ne sme biti kvalificiran s `const`; če je struktura ali unija, njene komponente, rekurzivno, ne smejo biti kvalificirane kot `const`. Tip prirejanja je tip levega operanda in vrednost prirejanja je vrednost levega operanda potem, ko se je prirejanje že zgodilo.

V enostavnem primeru z operatorjem `=`, vrednost izraza na desni nadomesti vrednost objekta, na katerega se *lvrednost* nanaša. Ena od naslednjih izjav mora biti pravilna: oba operanda sta aritmetičnega tipa in v tem primeru se desni operand s prirejanjem pretvori v tip levega operanda; oba operanda sta uniji ali strukturi istega tipa; en operand je kazalec, drugi je kazalec na `void`; levi operand je kazalec, desni pa konstanten izraz z vrednostjo `0`; oba operanda sta kazalca na funkcije ali objekte istih tipov z izjemo kvalifikacije `const` ali `volatile` v desnem operandu.

Izraz oblike `E1 op= E2` je ekvivalenten izrazu `E1 = E1 op (E2)` z razliko, da se `E1` izračuna samo enkrat.

B.7.18 Operator vejica

izraz:
    prirejanje
    izraz , *prirejanje*

Par izrazov, ločenih z vejico, se izračuna z leve na desno in vrednost levega izraza zavrže. Tip in vrednost rezultata sta tip in vrednost desnega izraza. Levi izraz se izračuna do kraja, z vsemi stranskimi učinki, preden se zavrže. V kontekstih, kjer ima vejica poseben pomen, na primer v seznamu funkcijskih argumentov (§B.7.3.2) in v seznamu inicializatorjev (§B.8.7), je zahtevana sintaktična enota prirejanje, tako da se vejica pojavi samo v posebnih oklepajih; na primer, klic funkcije f

```
f(a, (t = 3, t + 2), c)
```

ima tri argumente, od katerih ima drugi vrednost 5.

B.7.19 Konstantni izrazi

Sintaktično je konstanten izraz omejen na neko podmnožico operatorjev:

konstantni-izraz:
    pogojni-izraz

Izrazi, ki se zreducurajo na konstanto, so potrebni v raznih kontekstih: za ključno besedo `case`, število komponent večkratne vrednosti, širina bitnega polja, vrednost enumeracijskih konstant, v inicializatorjih in v nekaterih predprocesorjevih izrazih.

Konstantni izrazi ne morejo vsebovati prirejanj, operatorjev povečevanja in zmanjševanja, klicev funkcij in operatorja vejica, razen kot operandov `sizeof` operatorja. Če od konstantnega izraza zahtevamo, da je celoštevilčen, morajo biti njegovi operandi cela števila, enumeracije, znaki in realne konstante; prisile morajo predpisovati celoštevilčen tip in vse realne konstante morajo biti prisiljene v cela števila. S to zahtevo so izločene večkratne vrednosti, indirekcije, adresni in strukturni operatorji. (Kot operand operatorja `sizeof` lahko uporabimo poljuben operand).

Konstantni izrazi inicializatorjev dopuščajo več svobode: operandi so lahko konstante poljubnega tipa, unarni operator `&` lahko uporabimo na eksternih in na statičnih objektih in na eksternih in statičnih večkratnih vrednostih, indeksiranih s konstantnim izrazom. Unarni operator `&` lahko uporabimo tudi implicitno pri večkratni vrednosti brez indeksa in pri funkciji. Inicializatorji se morajo zreducirati na konstanto ali pa na naslov predhodno deklariranega statičnega ali eksternega objekta plus ali minus konstanta.

Manj svobode dopuščajo celoštevilčni konstantni izrazi, ki slede `#if`; `sizeof` izrazi, enumeracijske konstante in prisile niso dovoljene.

B.8 Deklaracije

Deklaracije določajo interpretacijo posameznih imen; praviloma ne rezervirajo pomnilnika za objekt povezan z imenom. Deklaracije, ki rezervirajo pomnilnik, se imenujejo definicije. Deklaracije imajo obliko

deklaracija:
  določila-deklaracije seznam-init-deklaratorjev$_{opt}$;

Deklaratorji v seznam-init-deklaratorjev vsebujejo imena, ki jih deklariramo; določila deklaracije so zaporedje določil tipa in pomnilniškega razreda.

določila-deklaracije:
  določilo-pomnilniškega-razreda določila-deklaracije$_{opt}$
  določilo-tipa določila-deklaracije$_{opt}$
  kvalifikator-tipa določila-deklaracije$_{opt}$

seznam-init-deklaratorjev:
  init-deklarator
  seznam-init-deklaratorjev , *init-deklarator*

init-deklarator:
  deklarator
  deklarator = *inicializator*

Deklaratorje bomo obravnavali kasneje (§B.8.5); vsebujejo imena, ki jih deklariramo. Deklaracija mora vsebovati vsaj en deklarator, ali pa mora njeno določilo tipa deklarirati značko strukture ali unije ali pa komponente enumeracije; prazne deklaracije niso dovoljene.

B.8.1 Določila pomnilniškega razreda

Določila pomnilniškega razreda so:

določilo-pomnilniškega-razreda:
  `auto`
  `register`
  `static`
  `extern`
  `typedef`

Pomen pomnilniškega razreda smo obravnavali v §B.4.

Določili `auto` in `register` dasta objektu avtomatičen pomnilniški razred in ju lahko uporabljamo samo znotraj funkcij. Take deklaracije služijo tudi kot definicije in povzroče, da dobe objekti rezerviran pomnilnik. Deklaracija `register` je ekvivalentna deklaraciji `auto` z namigom, da bo objekt pogosto uporabljan. Samo nekaj objektov je lahko dejansko shranjenih v registrih in samo določeni tipi pridejo v poštev; omejitve so odvisne od izvedbe. Če je objekt deklariran kot `register`, ne smemo zanj uporabiti operatorja `&`, niti eksplicitno niti implicitno.

Določilo `static` daje objektu statičen pomnilniški razred in ga lahko uporabimo bodisi znotraj funkcije bodisi zunaj nje. Znotraj funkcije to določilo povzroči, da prevajalnik dodeli pomnilnik, torej služi kot definicija; za učinek nunaj funkcije glej §B.11.2.

Določilo **`extern`**, uporabljeno znotraj funkcije, določa, da je pomnilnik za objekt definiran nekje drugje; za učinke zunaj funkcije glej §B.11.2.

Določilo **`typedef`** ne rezervira pomnilnika in se imenuje določilo pomnilniškega razreda samo iz udobja; obravnavamo ga v §B.8.9.

Deklaracija ima lahko kvečjemu eno določilo pomnilniškega razreda. Če ne uporabimo niti enega, stopijo v veljavo naslednja pravila: objekti, deklarirani znotraj funkcije, veljajo kot **`auto`**, funkcije, deklarirane znotraj funkcije, veljajo za **`extern`**; objekti in funkcije, deklarirani zunaj funkcij, veljajo za statične z eksterno vezavo. Glej §§B.10-§B.11.

B.8.2 Določila tipa

Določila tipa so

določilo-tipa:
`void`
`char`
`short`
`int`
`long`
`float`
`double`
`signed`
`unsigned`
določilo-strukture-ali-unije
določilo-enumeracije
typedef-ime

Kvečjemu eno od besed `long` in `short` lahko predpišemo skupaj z `int`; pomen je isti, če `int` ne omenjamo. Besedo `long` lahko predpišemo skupaj z `double`. Kvečjemu eno besedo od `signed` in `unsigned` lahko uporabimo skupaj z `int`, njegovo `short` ali `long` inačico ali s `char`. Beseda lahko stoji sama zase in v tem primeru se razume tip `int`. Določilo `signed` je koristno, ker z njim lahko prisilimo objekte tipa `char`, da nosijo s seboj predznak; določilo je dovoljeno, a odvečno, pri drugih celoštevilčnih tipih.

Na splošno lahko uporabimo v deklaraciji kvečjemu eno določilo tipa. Če v deklaraciji določila tipa ni, velja `int`.

Tipi so lahko tudi kvalificirani, da lahko naznačimo posebne lastnosti objektov, ki jih deklariramo.

kvalifikator-tipa:
`const`
`volatile`

Kvalifikatorje tipa lahko uporabimo skupaj z vsakim določilom tipa. Kvalifikacija `const` pomeni, da lahko objekt inicializiramo, potem ga pa ne smemo več spreminjati. Semantika kvalifikacije `volatile` je odvisna od izvedbe.

B.8.3 Deklaracije struktur in unij

Struktura je objekt, ki ga tvori zaporedje imenovanih objektov (v splošnem) različnih tipov. Unija je objekt, ki hrani, ob različnih časih, enega od več imenovanih objektov (v splošnem) različnih tipov. Določila struktur in unij imajo isto obliko.

določilo-strukture-ali-unije:
  struct-ali-union ime_{opt} {*seznam-struct-deklaracij* }
  struct-ali-union ime

struct-ali-union:
  `struct`
  `union`

Seznam struct deklaracij je zaporedje deklaracij za elemente strukture ali unije:

seznam-struct-deklaracij:
  struct-deklaracija
  seznam-struct-deklaracij struct-deklaracija

struct-deklaracija:
  spec-in-kval-seznam seznam-struct-deklaracij ;

spec-in-kval-seznam:
  določilo-tipa $spec\text{-}in\text{-}kval\text{-}seznam_{opt}$
  kvalifikator-tipa $spec\text{-}in\text{-}kval\text{-}seznam_{opt}$

seznam-struct-deklaracij:
  struct-deklarator
  seznam-struct-deklaracij , *struct-deklarator*

Običajno je struct-deklarator kar deklarator za komponento strukture ali unije. Komponenta strukture je lahko tudi zgrajena iz predpisanega števila bitov. Taki komponenti pravimo tudi bitno polje, ali samo polje; njegova širina je ločena od deklaratorja imena z dvopičjem.

struct-deklarator:
  deklarator
  $deklarator_{opt}$: *konstantni-izraz*

Določilo tipa oblike

```
struct-ali-union ime { seznam-struct-deklaracij }
```

deklarira ime kot značko strukture ali unije, določene s seznamom. Sledeče deklaracije v istem ali v vloženem dosegu se lahko sklicujejo na isti tip s tem, da uporabijo značko v določilu brez seznama:

```
struct-ali-union ime
```

Če se pojavi določilo z značko, a brez seznama, v situaciji, kjer značka ni deklarirana, je to nepopoln tip. Objekte z nepopolnim strukturnim ali unijskim tipom lahko uporabljamo v kontekstih, kjer ne potrebujemo njihove velikosti, na primer v deklaracijah (ne pa v definicijah), za določilo kazalca ali za kreiranje `typedef` deklaracije, sicer pa ne. Nepopoln tip postane popoln, ko se v nadaljevanju pojavi določilo z isto značko in seznamom struct deklaracij. Celo v določilih s seznamom je strukturni ali unijski tip definiran kot nepopoln znotraj seznama, popoln postane šele z `}`, ki zaključuje določilo.

Struktura ne sme vsebovati nobene komponente nepopolnega tipa. Zato je nemogoče deklarirati strukturo ali unijo, ki vsebuje samo sebe kot element. Po drugi strani, poleg tega, da poimenujejo strukturne in unijske tipe, značke omogočajo definicijo samoreferenčnih struktur; struktura ali unija lahko vsebuje kazalec na primerek same sebe, ker lahko deklariramo kazalce na nepopolne tipe.

Specialno pravilo se nanaša na deklaracije oblike

```
struct-ali-union ime ;
```

ki deklarirajo strukturo ali unijo, a nimajo ne seznama deklaracij niti deklaratorjev. Celo če je ime značka strukture ali unije že deklarirane v zunanjem dosegu (§B.11.1), taka deklaracija povzroči, da je ime značka novega nepopolnega strukturnega ali unijskega tipa v trenutnem dosegu.

Določilo strukture ali unije s seznamom komponent, a brez značke, ustvari nov tip; lahko ga uporabimo direktno samo v deklaraciji, katere del je.

Imena komponent in značk se ne tepejo niti med seboj niti z imeni običajnih spremenljivk. Ime komponente se ne sme pojaviti dvakrat v isti strukturi ali uniji, a isto ime lahko uporabljamo v različnih strukturah ali unijah.

Komponenta strukture ali unije, ki ni bitno polje, je poljubnega tipa. Bitno polje (, ki je lahko brez deklaratorja, torej neimenovano,) mora biti tipa `int`, `unsigned int` ali `signed int` in je interpretirano kot objekt celoštevilčnega tipa s predpisanim številom bitov; vprašanje, ali je `int` bitno polje predznačeno, je prepuščeno izvedbi. Zaporedna bitna polja v strukturi so zložena v pomnilniške enote, katerih velikost je odvisna od izvedbe, tako kot smer zlaganja. Kadar bitnega polja, ki sledi drugemu bitnemu polju, ne moremo več shraniti v delno napolnjeno pomnilniško enoto, ga prevajalnik lahko razdeli med dve pomnilniški enoti ali pa dopolni predhodno enoto in shrani polje na začetek nove enote. Neimenovano polje širine `0` bitov izsili tako poravnavanje tako, da se naslednje bitno polje začne na robu pomnilniške enote.

Komponente strukture imajo naslove v naraščajočem vrstem redu njihovih deklaracij. Komponenta, ki ni bitno polje, je poravnana na naslovno mejo, ki je odvisna od tipa; tako lahko nastanejo neimenovane luknje v strukturi. Če kazalec na strukturo prisilimo v kazalec na njeno prvo komponento, se rezultat nanaša na prvo komponento.

Unijo lahko razumemo kot strukturo, kjer so vse komponente na istem naslovu v pomnilniku in ki je dovolj široka, da hrani katerokoli komponento. V unijo lahko shranimo ob vsakem času samo eno komponento. Če kazalec na unijo prisilimo v kazalec na neko njeno komponento, se rezultat nanaša na to komponento.

Preprost primer deklaracije strukture je

```
struct tnode {
    char tword[20];
    int count;
    struct tnode *left;
    struct tnode *right;
};
```

ki vsebuje dvajsetkratni znak, celo število in dva kazalca na podobne strukture. Potem ko smo to deklaracijo zapisali, deklaracija

```
struct tnode s, *sp;
```

deklarira `s` kot strukturo dane vrste in `sp` kot kazalec na tako strukturo. V dosegu teh deklaracij se izraz

```
sp->count
```

nanaša na komponento `count` v strukturi, na katero sp kaže;

```
s.left
```

se nanaša na levo poddrevo v strukturi s; in

```
s.right->tword[0]
```

se nanaša na prvi znak komponente `tword` v desnem poddrevesu strukture `s`.

V splošnem, komponente unije ne smemo pogledati, če nismo priredili uniji take komponente. Posebno pravilo pa poenostavlja uporabo unij: če unija vsebuje nekaj struktur, ki imajo skupno začetno zaporedje komponent, in če unija trenutno vsebuje eno od teh struktur, tedaj je dovoljeno sklicevati se na skupni začetni del katerekoli od vsebovanih struktur. Tako je na primer naslednji odlomek legalen:

```
union {
    struct {
        int type;
    } n;
    struct {
        int type;
        int intnode;
    } ni;
    struct {
        int type;
        float floatnode;
    } nf;
} u;
...
u.nf.type = FLOAT;
u.nf.floatnode = 3.14;
...
if (u.n.type == FLOAT)
    ... sin(u.nf.floatnode) ...
```

B.8.4 Enumeracije

Enumeracije so posebni tipi z vrednostmi iz množice konstant z imeni, imenovanimi enumeratorji. Oblika določila enumeracije je izposojena pri strukturah in unijah.

določilo-enumeracije:
  `enum` ime_{opt} {*seznam-enumeratorjev* }
  `enum` *ime*

seznam-enumeratorjev:
enumerator
seznam-enumeratorjev , *enumerator*

enumerator:
ime
ime `=` *konstantni-izraz*

Imena v seznamu enumeratorjev so deklarirana kot konstante tipa `int` in se lahko pojavijo povsod, kjer je zahtevana konstanta. Če enumeratorjev z `=` ni, potem se vrednosti ustreznih konstant začnejo z `0` in povečujejo po `1`, če deklaracijo beremo z leve proti desni. Enumerator z `=` daje ustreznemu imenu predpisano vrednost; sledeča imena nadaljujejo z vrednostmi od prirejene dalje.

Imena enumeratorjev v nekem dosegu morajo biti vsa različna med seboj in različna od imen spremenljivk, vrednosti pa ni treba, da so med seboj različne.

Vloga imena v določilu enumeracije je podobna vlogi značke v določilu strukture ali unije; ime poimenuje izbrano enumeracijo. Pravila za določila enumeracije z značko ali brez nje in s seznamom enumeratorjev ali brez njega, so ista kot za določila strukture ali unije, z izjemo, da nepopolni enumeracijski tipi ne obstajajo; značka določila enumeracije brez seznama enumeratorjev se mora nanašati na določilo s seznamom, ki je v dosegu.

B.8.5 Deklaratorji

Deklaratorji imajo sintakso:

deklarator:
kazalec$_{opt}$ *direktni-deklarator*

direktni-deklarator:
ime
(*deklarator*)
direktni-deklarator [*konstantni-izraz*$_{opt}$]
direktni-deklarator (*seznam-tipov-parametrov*)
direktni-deklarator (*seznam-imen*$_{opt}$)

kazalec:
* *seznam-kvalifikatorjev-tipa*$_{opt}$
* *seznam-kvalifikatorjev-tipa*$_{opt}$ *kazalec*

seznam-kvalifikatorjev-tipa:
kvalifikator-tipa
seznam-kvalifikatorjev-tipa kvalifikator-tipa

Struktura deklaratorjev spominja na strukturo izrazov z indirekcijo, funkcijami in večkratnimi vrednostmi; asociiranje je isto.

B.8.6 Pomen deklaratorjev

Za zaporedjem deklaratorjev tipa in pomnilniškega razreda v deklaraciji stoji seznam deklaratorjev. Vsak deklarator deklarira enolično glavno ime, tisto, ki se pojavi kot prva alternativa produkcije za direktni-deklarator. Določila pomnilniškega razreda se nanašajo direktno na to ime, a tip je odvisen od oblike deklaratorja. Deklarator preberemo kot izjavo, da je ime, ki se pojavi v tako zgrajenem izrazu, predpisanega tipa.

Upoštevaje samo tipne dele določila deklaracije (§B.8.2) in izbrani deklarator, ima deklaracija obliko `T D`, kjer je `T` nek tip in `D` nek deklarator. Tip, ki pripada imenu v različnih oblikah deklaratorja, opišemo s takšnim označevanjem, rekurzivno.

V deklaraciji `T D`, kjer je `D` neko neokrašeno ime, je tip imena `T`.

V deklaraciji `T D`, kjer ima `D` obliko

```
( D1 )
```

je tip imena v `D1` isti kot v `D`. Oklepaji ne spremene tipa, lahko pa spremene asociacijo v kompleksnih deklaratorjih.

B.8.6.1 Deklaratorji kazalcev

V deklaraciji `T D`, kjer je `D` oblike

`* seznam-kvalifikatorjev-tipa`$_{opt}$ `D1`

in je ime v deklaraciji `T D1` tipa "modifikator-tipa `T`", je tip imena v `D` "modifikator-tipa seznam-kvalifikatorjev-tipa kazalec na `T`". Kvalifikatorji, ki slede `*`, se nanašajo na kazalec namesto na objekt, na katerega kazalec kaže.

Za primer si oglejmo deklaracijo

```
int *ap[];
```

Tu igra `ap[]` vlogo `D1`; deklaracija `int ap[]` (spodaj) bi povzročila, da bi bilo ime `ap` tipa "večkratni `int`", seznam kvalifikatorjev tipa je prazen, modifikator-tipa je "večkratni". Po dejanski deklaraciji je `ap` tipa "večkratni kazalec na `int`".

Kot dodatni primeri, deklaracije

```
int i, *pi, *const cpi = &i;
const int ci = 3, *pci;
```

deklarirajo `i` kot `int`, in `pi` kot kazalec na `int`. Vrednosti konstantnega kazalca `cpi` ne moremo spreminjati; vedno bo kazal na isto mesto v pomnilniku, vrednost, na katero kaže, pa lahko spreminjamo. Število `ci` je konstanta, ki je ne moremo spreminjati (čeprav jo lahko inicializiramo, kot v primeru). Kazalec `pci` je "kazalec na konstanten `int`" in `pci` sam smemo spremeniti, da pokaže nekam drugam, vrednosti, na katero pa nek trenutek kaže, ne moremo spreminjati z uporabo kazalca `pci`.

B.8.6.2 Deklaratorji večkratnih vrednosti

V deklaraciji `T D`, kjer ima `D` obliko

`D1 [ konstantni-izraz`$_{opt}$ `]`

in je ime v deklaraciji `T D1` tipa "modifikator-tipa `T`", je tip imena v `D` "modifikator-tipa večkratni `T`". Če je konstani-izraz prisoten, mora biti celoštevilčnega tipa z vrednostjo večjo od `0`. Če konstantnega izraza, ki določa meje, ni, je večkratna vrednost nepopolnega tipa.

Večkratno vrednost lahko skonstruiramo iz aritmetičnih tipov, iz kazalcev, iz struktur, iz unij ali iz drugih večkratnih vrednosti (, da dobimo večdimenzionalno večkratno vrednost). Vsak tip, iz katerega gradimo večkratno vrednost, mora biti popoln; ne sme biti večkratna vrednost ali struktura nepopolnega tipa. To pomeni, da pri večdimenzionalni večkratni vrednosti lahko opustimo samo prvo dimenzijo. Tip objekta nepopolne večkratne vrednosti postane popoln, ko naletimo na drugo, popolno, deklaracijo objekta (§B.10.2) ali pa da objekt inicializiramo (§B.8.7). Na primer,

```
float fa[17], *afp[17];
```

deklarira večkraten `float` in večkratni kazalec na `float`. Dalje,

```
static int x3d[3][5][7];
```

deklarira tridimenzionalno večkratno celoštevilčno vrednost ranga `3x5x7`. V podrobnostih, `x3d` je trikratna vrednost; vsaka od njih je petkratna vrednost; vsaka od njih je sedemkratni `int`. Vsi izrazi `x3d`, `x3d[i]`, `x3d[i][j]`, `x3d[i][j][k]` se lahko pojavijo v kakšnem izrazu. Prvi trije so večkratne vrednosti, zadnji je tipa `int`. Bolj podrobno, `x3d[i][j]` je sedemkratni `int` in `x3d[i]` je petkratni (sedemkratni `int`).

Operacija indeksiranja je definirana tako, da je `E1[E2]` identično z `*(E1+E2)`. Zato je navkljub asimetričnemu videzu indeksiranje komutativna operacija. Zaradi pravil pretvarjanja, ki veljajo za `+` in za večkratne vrednosti (§§B.6.6, §B.7.1 in B.7.7), če je `E1` večkratna vrednost in `E2` celo število, se `E1[E2]` nanaša na `E2`-to komponento vrednosti `E1`.

V zgornjem primeru je `x3d[i][j][k]` ekvivalentno `*(x3d[i][j] + k)`. Prvi podizraz `x3d[i][j]` se pretvori po §B.7.1 v "kazalec na večkratno celo število"; po §B.7.7 seštevanje vključuje množenje s širino celega števila. Iz teh pravil sledi, da je večkratna vrednost shranjena po vrsticah (zadnji indeks variira najhitreje) in da prva meja v deklaraciji pomaga pri izračunu količine pomnilnika, ki ga večkratna vrednost zaseda, sicer pa ne sodeluje pri nadaljnjem računu.

B.8.6.3 Deklaratorji funkcij

V deklaraciji funkcije v novem stilu, `T D`, kjer ima `D` obliko

```
D1(seznam-tipov-parametrov)
```

in je ime v deklaraciji `T D1` tipa "modifikator-tipa `T`", je tip imena v `D` "modifikator-tipa funkcija z argumenti seznam-tipov-parametrov, ki vrača `T`".

Sintaksa parametrov je

seznam-tipov-parametrov:
    seznam-parametrov
    seznam-parametrov , ...

seznam-parametrov:
    deklaracija-parametra
    seznam-parametrov , *deklaracija-parametra*

deklaracija-parametra:
    določila-deklaracije deklarator
    določila-deklaracije abstraktni-deklarator$_{opt}$

V deklaraciji v novem stilu seznam parametrov določa tipe parametrov. Kot poseben primer deklaratorja funkcije v novem stilu je deklarator funkcije brez parametrov, kjer za seznam parametrov navedemo samo ključno besedo `void`. Če se seznam tipov parametrov konča z vejico in tropičjem, tedaj lahko funkcija sprejme več argumentov, kot je število eksplicitno opisanih parametrov; glej §B.7.3.2.

Tipi parametrov, ki so večkratne vrednosti ali funkcije, se spremene v kazalce, v soglasju s pravili za pretvarjanje parametrov; glej §B.10.1. Edino določilo pomnilniškega razreda, ki je dovoljeno v določilu deklaracije parametra, je določilo `register` in še to določilo se ignorira, razen če deklarator funkcije ne stoji v glavi definicije funkcije. Podobno, če deklaratorji v deklaracijah parametrov vsebujejo imena in deklarator funkcije ne stoji v glavi definicije funkcije, se taka imena takoj izgube. Abstraktne deklaratorje, ki imen ne omenjajo, obravnavamo v §B.8.8.

V deklaraciji funkcije v starem stilu, `T D`, kjer ima `D` obliko

$$\texttt{D1 (seznam-imen}_{opt}\texttt{)}$$

in je ime v deklaraciji `T D1` tipa "modifikator-tipa `T`", je tip imena v `D` "modifikator-tipa funkcija z nepredpisanimi argumenti, ki vrača `T`". Parametri, če so prisotni, imajo obliko

seznam-imen:
    ime
    seznam-imen , *ime*

V deklaratorju v starem stilu ne sme biti seznama imen, razen če deklarator funkcije stoji v glavi definicije funkcije (§B.10.1). Deklaracija ne nosi s seboj nobene informacije o tipih parametrov.

Na primer, deklaracija

```
int f(), *fpi(), (*pfi)();
```

deklarira `f` kot funkcijo, ki vrača celo število, `fpi` kot funkcijo, ki vrača kazalec na celo število in `pfi` kot kazalec na funkcijo, ki vrača celo število. V nobeni funkciji parametri niso predpisani; deklaracija je v starem stilu.

V deklaraciji v novem stilu

```
int strcpy(char *dest, const char *source), rand(void);
```

je `strcpy` funkcija, ki vrača `int`, ima dva argumenta, prvi je kazalec na `char`, drugi je kazalec na konstanten `char`. Imena parametrov so v bistvu komentarji. Druga funkcija nima argumentov in vrača `int`.

B.8.7 Inicializacija

Kadar deklariramo nek objekt, lahko njegov init-deklarator določa začetno vrednost imena, ki ga deklariramo. Inicializator, pred katerim stoji =, je bodisi izraz bodisi seznam inicializatorjev gnezden v zavitih oklepajih. Seznam se lahko konča z vejico, lepotilom za lično formatiranje.

inicializator:
    prirejanje
    {*seznam-inicializatorjev* }
    {*seznam-inicializatorjev* , }

seznam-inicializatorjev:
    inicializator
    seznam-inicializatorjev , *inicializator*

Vsi izrazi v inicializatorju statičnega objekta ali večkratne vrednosti morajo biti konstantni izrazi kot so opisani v §B.7.19. Izrazi v inicializatorju za `auto` ali `register` objekt ali za večkratno vrednost morajo biti pravtako konstantni izrazi, če je inicializator vklenjen v zavite oklepaje. Seveda, če je inicializator avtomatičnega objekta en sam izraz, ni treba, da je konstanten, a biti mora primernega tipa za prirejanje objektu.

Statičen objekt, ki ni eksplicitno inicializiran, je inicializiran kot da bi bil (ali njegove komponente) inicializiran na konstanto `0`, natančneje, vsi njegovi biti so inicializirani na nič. Začetna vrednost avtomatičnega objekta, ki ni eksplicitno inicializiran, je nedefinirana.

Inicializator kazalca ali objekta aritmetičnega tipa je en sam izraz, morda v zavitih oklepajih. Vrednost izraza je prirejena objektu.

Inicializator strukture je bodisi izraz istega tipa bodisi seznam inicializatorjev v zavitih oklepajih. Seznam ustreza posameznim komponentam strukture, vključno z vrstnim redom. Če je inicializatorjev manj kot komponent, so preostale komponente inicializirane na vse bite enake nič.

Kot poseben primer lahko inicializiramo večkraten znak z nizom; zaporedni znaki niza inicializirajo zaporedne komponente večkratne vrednosti. Podobno lahko širok niz inicializira večkraten `wchar_t`. Če ima večkraten znak nedoločeno velikost, število znakov v nizu, skupaj s končnim praznim znakom, določi njegovo velikost; če je velikost že predpisana, stevilo znakov v nizu, ne števši končni znak, ne sme presegati predpisane velikosti.

Inicializator unije je bodisi izraz istega tipa bodisi inicializator prve komponente unije v zavitih oklepajih.

Agregat je struktura ali večkratna vrednost. Če agregat vsebuje komponente agregatnih tipov, veljajo pravila inicializacije rekurzivno. Zavite oklepaje lahko v inicializaciji črtamo takole: če se inicializator komponente agregata, ki je sama agregat, začenja z odprtim zavitim oklepajem, potem sledeči seznam inicializatorjev, ločenih z vejico, inicializira komponente podagregata; če je inicializatorjev več kot komponent, je to napaka. Seveda, če se inicializator podagregata ne začenja z zavitim oklepajem, tedaj prevajalnik s seznama inicializatorjev pobere samo toliko inicializatorjev, kolikor jih je potrebnih za podagregat; preostali inicializatorji se porabijo za inicializacijo naslednje komponente agregata, katerega del je podagregat.

Na primer,

```
int x[] = { 1, 3, 5 };
```

deklarira in inicializira enodimenzionalno trikratno vrednost, ker velikost ni predpisana, imamo pa tri inicializatorje.

```
float y[4][3] = {
    { 1, 3, 5 },
    { 2, 4, 6 },
    { 3, 5, 7 },
};
```

je inicializacija z vsemi odvečnimi oklepaji: `1`, `3` in `5` inicializirajo prvo vrstico večkratne vrednosti `y[0]`, namreč `y[0][0]`, `y[0][1]` in `y[0][2]`. Podobno naslednji dve vrstici inicializirata `y[1]` in `y[2]`. Inicializator se predčasno konča, zato so elementi `y[3]` inicializirani na `0`. Natanko isti učinek dosežemo z

```
float y[4][3] = {
    1, 3, 5, 2, 4, 6, 3, 5, 7
};
```

Inicializator za `y` se začenja z odprtim zavitim oklepajem, inicializator za `y[0]` pa ne; zato porabi `y[0]` tri elemente s seznama. Podobno pobere naslednje tri elemente `y[1]` in nato še `y[2]`. Podobno

```
float y[4][3] = {
    { 1 }, { 2 }, { 3 }, { 4 }
};
```

inicializira prvi stolpec `y` (tolmačen kot dvodimenzionalna večkratna vrednost) in pusti ostale elemente na `0`.

Končno,

```
char msg[] = "Syntax error on line %s\n";
```

prikazuje večkraten znak, katerega komponente so inicializirane z nizom; velikost vključuje prazen znak.

B.8.8 Imena tipov

V nekaterih kontekstih (, da eksplicitno določimo tip pretvorbe s prisilo, da deklariramo tipe parametrov v deklaratorju funkcije, kot argument operatorja `sizeof`) je treba povedati ime podatkovnega tipa. To dosežemo z imenom tipa, ki je sintaktično deklaracija objekta takega tipa brez imena objekta.

ime-tipa:
    spec-in-kval-seznam abstraktni-deklarator$_{opt}$

abstraktni-deklarator:
    kazalec
    kazalec$_{opt}$ *direktni-abstraktni-deklarator*

direktni-abstraktni-deklarator:
    (*abstraktni-deklarator*)
    direktni-abstraktni-deklarator$_{opt}$ [*konstantni-izraz*$_{opt}$]
    direktni-abstraktni-deklarator$_{opt}$ (*seznam-tipov-parametrov*$_{opt}$

Enolično je možno določiti položaj v abstraktnem deklaratorju, kjer bi napisali ime, če bi bila konstrukcija deklarator v deklaraciji. Ime tipa je potem isto kot tip hipotetičnega imena. Na primer,

```
int
int *
int *[3]
int (*)[]
int *()
int (*[])(void)
```

imenujejo, po vrsti, tipe "celo število", "kazalec na celo število", "trikratni kazalec na celo število", "kazalec na večkratno celo število", "funkcija z nepredpisanimi parametri, ki vrne kazalec na celo število" in "večkraten kazalec na funkcijo brez parametrov, ki vrne celo število".

B.8.9 Deklaracija typedef

Deklaracije z določilom pomnilniškega razreda `typedef` ne deklarirajo objektov, ampak imena, ki imenujejo tip. Ta imena se imenujejo `typedef` imena.

typedef-ime:
  ime

Deklaracija `typedef` pripiše tip vsekemu imenu med njenimi deklaratorji na običajen način (glej §B.8.6). V nadaljevanju je vsako tako `typedef` ime sintaktično enakovredno ključni besedi določila tipa za pridruženi tip.

Na primer, po deklaraciji

```
typedef long Blockno, *Blockptr;
typedef struct { double r, theta; } Complex;
```

so konstrukcije

```
Blockno b;
extern Blockptr bp;
Complex z, *zp;
```

legalne deklaracije. Tip spremenljivke `b` je `long`, tip `bp` je "kazalec na `long`" in tip spremenljivke `z` je predpisana struktura; `zp` je kazalec na tako strukturo.

Deklaracija `typedef` ne vpelje novih tipov, le sinonime za tipe, ki jih lahko določimo drugače. V primeru zgoraj je `b` istega tipa kot vsak drug `long` objekt.

Imena, deklarirana s `typedef` deklaracijo, lahko ponovno deklariramo v notranjem dosegu, a uporabiti moramo neprazno množico določil tipa. Na primer,

```
extern Blockno;
```

ne deklarira ponovno Blockno, a

```
extern int Blockno;
```

ga deklarira ponovno.

B.8.10 Ekvivalenca tipov

Dva seznama določil tipa sta ekvivalentna, če vsebujeta isti nabor določil tipa in če pri tem upoštevamo, da lahko nekatera določila tipa impliciramo iz drugih (na primer, `long` sam zase implicira `long int`). Strukture, unije in enumeracije z različnimi značkami so različne in struktura, unija ali enumeracija brez značke pomeni nov tip.

Dva tipa sta enaka, če sta njuna abstraktna deklaratorja (§B.8.8), po substituciji `typedef` tipov in po brisanju imen parametrov funkcij, ista do ekvivalence seznamov določil tipa. Število komponent v večkratni vrednosti in tipi parametrov funkcij so pri tem pomembni.

B.9 Stavki

Če ni posebej rečeno drugače, se stavki izvršujejo zapored. Stavki se izvajajo zaradi svojega učinka in nimajo vrednosti. Delimo jih v več skupin.

stavek:
    označeni-stavek
    izrazni-stavek
    sestavljeni-stavek
    izbirni-stavek
    iteracijski-stavek
    skočni-stavek

B.9.1 Označeni stavki

Stavki lahko nosijo oznako kot predpono.

označeni-stavek:
    ime : *stavek*
    `case` *konstantni-izraz* : *stavek*
    `default` : *stavek*

Oznaka, ki jo tvori ime, deklarira to ime. Edina uporaba imena oznake je kot tarča stavka `goto`. Doseg imena je trenutna funkcija. Ker imajo imena oznak svoj prostor imen, se ne tepejo z drugimi imeni in jih ne moremo ponovno deklarirati. Glej §B.11.1.

Oznake vrste `case` in `default` uporabljamo v stavku `switch` (§B.9.4). Konstantni izraz v `case` oznaki mora biti celoštevilčnega tipa.

Oznake same po sebi ne spremene poteka programa.

B.9.2 Izrazni stavek

Večina stavkov so izrazni stavki, ki imajo obliko

izrazni-stavek:
    izraz$_{opt}$;

Večina izraznih stavkov v povprečnem programu so prirejanja ali klici funkcij. Vsi stranski učinki izraza se dopolnijo, preden se izvede naslednji stavek. Če izraza v izraznem stavku ni, je to prazen stavek; često ga uporabljamo, da zapišemo prazno telo iteracijskega stavka ali pa da zapišemo predenj oznako.

B.9.3 Sestavljeni stavek

Sestavljeni stavek (imenovan tudi ”blok”) uporabljamo tam, kjer prevajalnik pričakuje en sam stavek, potrebujemo jih pa več. Telo funkcijske definicije je sestavljeni stavek.

sestavljeni-stavek:
    {*seznam-deklaracij*$_{opt}$ *seznam-stavkov*$_{opt}$ }

seznam-deklaracij:
    deklaracija
    seznam-deklaracij deklaracija

seznam-stavkov:
    stavek
    seznam-stavkov stavek

Če je bilo neko ime v seznamu deklaracij v dosegu zunaj bloka, je zunanja deklaracija znotraj bloka zastrta (glej §B.11.1), po koncu bloka pa spet stopi v veljavo. Ime je v nekem bloku lahko deklarirano samo enkrat. Ta pravila se nanašajo na imena v istem prostoru imen (§B.11); imena v različnih prostorih imen se tretirajo kot različna imena.

Inicializacija avtomatičnih objektov se zgodi vsakokrat, ko vstopimo na vrhu v blok in se izvaja v takem vrstnem redu kot so deklaratorji napisani. Če izvedemo skok v notranjost bloka, se te inicializacije ne zgode. Inicializacija statičnih objektov se zgodi samo enkrat, v principu preden se program začne izvajati.

B.9.4 Izbirni stavek

Izbirni stavki izbirajo med več tokovi kontrole.

izbirni-stavek:
    `if (` *izraz* `)` *stavek*
    `if (` *izraz* `)` *stavek* `else` *stavek*
    `switch (` *izraz* `)` *stavek*

V obeh oblikah stavka `if` se izraz, ki mora biti aritmetičnega ali kazalčnega tipa, izračuna, vključno z vsemi stranskimi učinki, in če je vrednost različna od `0`, se izvrši prvi podstavek. V drugi obliki se izvrši drugi podstavek, če je vrednost izraza enaka `0`. Dvoumen `else` se razreši tako, da `else` pripada zadnjemu stavku `if` brez `else` v istem bloku.

Stavek `switch` prenese kontrolo izvajanja na enega od več stavkov, odvisno od vrednosti izraza, ki mora biti celoštevilčnega tipa. Podstavek, ki ga kontrolira `switch`, je tipično sestavljeni stavek. Vsak stavek v podstavku je lahko označen z eno ali večimi `case` oznakami (§B.9.1). Kontrolni izraz pretrpi celoštevilčno promocijo (§B.6.1) in `case` konstante se pretvorijo v promoviran tip. Nobeni dve `case` konstanti, ki pripadata istemu `switch` stavku, ne smeta po pretvarjanju imeti iste vrednosti. V vsakem stavku `switch` je ravno tako lahko le ena oznaka `default`. Stavki `switch` so lahko gnezdeni; `case` ali `default` oznaka pripada najmanšemu `switch` stavku, ki jo vsebuje.

Ko se `switch` stavek izvaja, se izraz izračuna, vključno z vsemi stranskimi učinki, vrednost izraza pa se primerja z vsako `case` konstanto. Če je neka `case` konstanta enaka vrednosti izraza, kontrolo prevzame stavek, označen z ustrezno oznako. Če nobena `case` konstanta ni enaka vrednosti izraza in če obstaja `default` oznaka, kontrolo prevzame stavek označen z `default` oznako. Če nobena `case` oznaka ni enaka vrednosti izraza in če ni `default` oznake, se ne izrši noben podstavek.

B.9.5 Iteracijski stavki

Z iteracijskimi stavki tvorimo zanke.

iteracijski stavek:
    `while (` *izraz* `)` *stavek*
    `do` *stavek* `while (` *izraz* `) ;`
    `for (` $izraz_{opt}$ `;` $izraz_{opt}$ `;` $izraz_{opt}$ `)` *stavek*

V stavkih `while` in `do` se podstavek izvaja toliko časa dokler je vrednost izraza različna od `0`; `izraz` mora biti aritmetičnega ali kazalčnega tipa. V stavku `while` se test, skupaj z vsemi stranskimi učinki, zgodi pred vsakim izvajanjem stavka; v zanki `do` test sledi vsaki iteraciji.

V stavku `for` se prvi `izraz` izračuna enkrat in tako predstavlja inicializacijo zanke. Na tip izraza ni nobene omejitve. Drugi `izraz` mora biti aritmetičnega ali kazalčnega tipa; izračuna se pred vsako iteracijo in če je enak `0`, je zanka `for` končana. Tretji `izraz` se izračuna po vsaki iteraciji in tako predstavlja reinicializacijo zanke. Na tip izraza ni nobene omejitve. Stranski učinki vsakega izraza se dopolnijo takoj po njegovem izračunu. Če podstavek ne vsebuje stavka `continue`, je stavek

```
for ( izraz1 ; izraz2 ; izraz3 ) stavek
```

ekvivalenten konstrukciji

```
izraz1;
while ( izraz2 ) {
    stavek
    izraz3 ;
}
```

Vsakega od treh izrazov lahko opustimo. Manjkajoč drugi izraz povzroči, da je implicitni test ekvivalenten testiranju neničelne konstante.

B.9.6 Skočni stavek

Skočni stavki brezpogojno prenesejo kontrolo.

skočni-stavek:
`goto` *ime* ;
`continue` ;
`break` ;
`return` *izraz*$_{opt}$;

V stavku `goto` mora biti `ime` oznaka (§B.9.1), ki se nahaja v trenutni funkciji. Kontrola se prenese na označeni stavek.

Stavek `continue` se lahko pojavi samo v iteracijskem stavku in povzroči, da se kontrola preda nadaljevalnemu delu najmanjše zanke, ki tak stavek vsebuje. Bolj natančno, v vsakem od stavkov

```
while (...) {         do {                  for (...) {
    ...                   ...                   ...
contin: ;             contin: ;             contin: ;
}                     } while (...);        }
```

je `continue`, ki ni vsebovan v manjšem iteracijskem stavku, isto kot `goto` contin.

Stavek `break` se lahko pojavi samo v iteracijskem ali `switch` stavku in prekine izvajanje najmanjšega takega stavka; kontrola se nadaljuje na prvem stavku, ki sledi prekinjenemu stavku.

Funkcija preda kontrolo svojemu klicatelju s stavkom `return`. Če besedi `return` sledi izraz, dobi klicatelj to vrednost kot vrednost funkcije. Tip izraza se pretvori, kot pri prirejanju, v tip, ki ga funkcija, v kateri se `return` nahaja, vrača.

Če se funkcija izteče v zaključni zaviti oklepaj, je to ekvivalentno stavku `return` brez izraza. V obeh primerih je vrnjena vrednost nedefinirana.

B.10 Eksterne deklaracije

Enota podatkov, ki jih dobi C prevajalnik, se imenuje prevajalna enota; tvori jo zaporedje eksternih deklaracij, ki so bodisi deklaracije bodisi definicije funkcij.

prevajalna-enota:
eksterna-deklaracija prevajalna-enota eksterna-deklaracija

eksterna-deklaracija:
definicija-funkcije deklaracija

Doseg eksternih deklaracij se razteza do konca prevajalne enote, v kateri so deklarirane, tako kot se učinek deklaracije v bloku razteza do konca bloka. Sintaksa eksternih deklaracij je ista kot za vse deklaracije, razen da samo na tem nivoju lahko podamo kod funkcije.

B.10.1 Definicije funkcij

Definicije funkcij imajo obliko

definicija-funkcije:
določilo-deklaracije$_{opt}$ *deklarator seznam-deklaracij*$_{opt}$ *jedro*

jedro:
sestavljeni-stavek

Edini določili pomnilniškega razreda, ki sta dovoljena med določili deklaracije, sta `extern` in `static`; glej §B.11.2 za razliko med njima.

Funkcija lahko vrne aritmetičen tip, strukturo, unijo, kazalec ali `void`, ne more pa vrniti funkcije ali večkratne vrednosti. Deklarator v deklaraciji funkcije mora eksplicitno določati, da deklarira funkcijski tip; to je, vsebovati mora eno od oblik (glej §B.8.6.3)

direktni-deklarator (*seznam-tipov-parametrov*)
direktni-deklarator (*seznam-imen*$_{opt}$)

kjer je *direktni-deklarator* ime ali ime v oklepajih. Specialno, direktni deklarator ne more postati funkcijski deklarator na račun `typedef`.

V prvi obliki je definicija v novem stilu in njeni parametri, skupaj s svojimi tipi, so deklarirani v njenem seznamu tipov parametrov; funkcijskemu deklaratorju ne sme slediti seznam deklaracij. Razen če je seznam tipov parametrov tvorjen samo iz `void`, ki pomeni, da funkcija nima parametrov, mora vsak deklarator v seznamu tipov parametrov vsebovati neko ime. Če se seznam tipov parametrov konča z ”`, ...`”, lahko funkcijo pokličemo z več argumenti, kot ima parametrov; `va_arg` makro mehanizem, definiran v standardni glavi `<stdarg.h>` in opisan v dodatku C, moramo uporabiti za dostop do dodatnih argumentov. Variadične funkcije morajo imeti vsaj en imenovan parameter.

V drugi obliki je definicija v starem stilu; seznam imen našteje parametre, seznam deklaracij jim pripiše tipe. Če za dani parameter ni deklaracije, je njegov tip `int`. Seznam deklaracij mora deklarirati samo parametre, naštete v seznamu, inicializacija ni dovoljena in edino določilo pomnilniškega razreda, ki je dovoljeno, je `register`.

V obeh stilih definicij funkcij se razume, da so parametri deklarirani takoj na začetku sestavljenega stavka, ki tvori telo funkcije, in tam ne morejo biti ponovno deklarirani (čeprav so lahko, tako kot druga imena, ponovno deklarirani v notranjem bloku). Če je parameter deklariran kot "večkratni tip", se deklaracija predela v "kazalec na tip"; podobno, če je parameter deklariran kot "funkcija, ki vrne tip", se deklaracija predela v "kazalec na funkcijo, ki vrne tip". Med klicem funkcije se argumenti pretvorijo, kot je to potrebno, in priredijo parametrom; glej §B.7.3.2.

Popoln primer definicije funkcije v novem stilu je

```
int max(int a, int b, int c)
{
    int m;

    m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

Tu je `int` določilo deklaracije;

```
max(int a, int b, int c)
```

je deklarator funkcije in `{...}` je blok s kodom funkcije. Ustrezna definicija v starem stilu bi bila

```
int max(a, b, c)
int a, b, c;
{
    int m;

    m = (a > b) ? a : b;
    return (m > c) ? m : c;
}
```

kjer je sedaj

```
int max(a, b, c)
```

deklarator in

```
int a, b, c;
```

je seznam deklaracij za parametre.

B.10.2 Eksterne deklaracije

Eksterne deklaracije določajo karakteristike objektov, funkcij in drugih imen. Izraz "eksteren" se nanaša na njihovo lego zunaj funkcij in ni direktno povezan s ključno besedo `extern`; pomnilniški razred objekta, ki je deklariran eksterno, lahko pustimo prazen, ali pa ga določimo kot `extern` ali `static`.

V eni prevajalni enoti lahko obstaja več eksternih deklaracij za isto ime, če se ujemajo v tipu in vezavi in če je kvečjemu ena od njih definicija imena.

Dve deklaraciji objekta ali funkcije se ujemata v tipu po pravilih §B.8.10. Poleg tega, če se deklaraciji ne ujemata v tipu, ker je en tip nepopoln strukturni, unijski ali enumeracijski tip (§B.8.3), drugi pa je ustrezen popoln tip z isto značko, potem pravimo, da se tipa ujemata. Še več, če je en tip nepopolna večkratna vrednost drugi pa popolna večkratna vrednost, tipa pa sta sicer enaka, ju tretiramo, kot da se ujemata. Končno, če en tip določa funkcijo v starem stilu, drugi pa sicer identično funkcijo v novem stilu, z deklaracijami parametrov, smatramo, da se tipa ujemata.

Če prva eksterna deklaracija funkcije ali objekta vsebuje določilo `static`, ima ime interno vezavo; sicer ima ime eksterno vezavo. Vezavo obravnavamo v §B.11.2.

Eksterna deklaracija objekta je definicija, če ima inicializator. Eksterna deklaracija objekta, ki nima inicializatorja in ki ne vsebuje določila `extern`, je provizorična definicija. Če se v katerikoli prevajalni enoti pojavi definicija objekta, tedaj vse provizorične definicije postanejo odvečne deklaracije. Če v nobeni prevajalni enoti ni definicije objekta, tedaj vse provizorične definicije skupaj postanejo ena sama definicija z inicializatorjem `0`.

Vsak objekt mora imeti natančno eno definicijo. Za objekte z interno vezavo velja to pravilo ločeno za vsako prevajalno enoto, ker ima vsaka prevajalna enota en sam interno povezan objekt z danim imenom. Za objekte z zunanjo vezavo velja to pravilo za ves program.

B.11 Doseg in vezava

Program ni treba da je preveden ves naenkrat: izvorni tekst lahko hranimo v večih datotekah, ki vsebujejo prevajalne enote, in vnaprej prevedene podprograme lahko naložimo iz knjižnic. Komuniciranje med funkcijami programa poteka s pomočjo klicev funkcij in s pomočjo manipulacij eksternih podatkov.

Zategadelj je treba obravnavati dve vrsti dosegov: najprej, leksični doseg imena je področje programskega teksta, kjer računalnik razume karakteristike imena; in nato doseg, povezan z objekti in funkcijami z eksterno vezavo, ki določa vezave med imeni v ločeno prevedenih prevajalnih enotah.

B.11.1 Leksični doseg

Imena spadajo v različne prostore imen, ki se ne tepejo med seboj; isto ime lahko uporabimo v drugačen namen, celo v istem dosegu, če se uporabe nanašajo na različne prostore imen. Ti razredi so: objekti, funkcije, `typedef` imena in `enum` konstante; oznake; značke struktur, unij in enumeracij; komponente vsake strukture ali unije posebej.

Leksični doseg imena objekta ali funkcije v eksterni deklaraciji se začenja na koncu njegovega deklaratorja in se razteza do konca prevajalne enote, v kateri se pojavlja. Doseg parametrov v definiciji funkcije se začenja na začetku bloka, ki definira funkcijo, in se razteza do konca funkcije; doseg parametra v deklaraciji funkcije se konča na koncu deklaratorja. Doseg imena, deklariranega v glavi bloka, se začenja na koncu njegovega deklaratorja in se razteza do konca bloka. Doseg oznake je vsa funkcija, kjer se oznaka pojavi. Doseg značke strukture, unije ali enumeracije ali enumeracijske konstante se začenja, ko se pojavi v določilu tipa in se razteza do konca prevajalne enote (za deklaracije na eksternem nivoju) ali do konca bloka (za deklaracije znotraj funkcije).

Če je neko ime eksplicitno deklarirano v glavi kakšnega bloka, vključno z blokom, ki tvori funkcijo, je vsaka deklaracija tega imena zunaj bloka zastrta do konca bloka.

B.11.2 Vezava

Znotraj prevajalne enote se vse deklaracije istega imena objekta ali funkcije z interno vezavo nanašajo na isto stvar in v prevajalni enoti je en sam objekt ali funkcija s tem imenom. Vse deklaracije istega imena objekta ali funkcije z eksterno vezavo se nanašajo na isto stvar in tak objekt ali funkcija sta skupna celemu programu.

Kot je bilo rečeno v §B.10.2, prva eksterna deklaracija imena daje imenu interno vezavo, če uporabimo določilo `static`, sicer pa eksterno vezavo. Če deklaracija imena znotraj nekega bloka ne vsebuje določila `extern`, ime nima vezave in je edinstveno v funkciji. Če deklaracija vsebuje določilo `extern` in je eksterna deklaracija imena vidna v dosegu, ki zajema blok, ima ime isto vezavo kot eksterna deklaracija in se nanaša na isti objekt ali funkcijo; če pa eksterna deklaracija ni vidna, je vezava eksterna.

B.12 Predprocesiranje

Predprocesor opravlja makro substitucije, pogojno prevajanje in vključevanje imenovanih datotek. Vrstice, ki se začenjajo z višajem `#`, pred katerim je lahko še nekaj belih znakov, komunicirajo s tem predprocesorjem. Sintaksa teh vrstic je neodvisna od preostanka jezika; lahko se pojavijo kjerkoli in imajo učinek, ki traja (neodvisno od dosega), do konca prevajalne enote. Meje med vrsticami so pomembne; vsaka vrstica je individualno analizirana (a glej §B.12.2 kako stikamo vrstice). Za predprocesor je leksični element katerikoli element jezika `C`, ali pa niz znakov, ki definira ime datoteke, na primer v `#include` direktivi (§B.12.4); poleg tega velja vsak znak, ki ni drugače definiran, za leksični element. Učinek belih znakov, drugih kot presledek in horizontalni predelčnik, je v predprocesorjevih vrsticah nedefiniran.

Predprocesiranje se izvaja v nekaj logično zaporednih fazah, ki jih sme posamezna izvedba kondenzirati.

1. Najprej, predprocesor nadomesti trigrafična zaporedja, kot so definirana v §B.12.1, z njihovimi ekvivalenti. Če operacijski sistem tako zahteva, predprocesor vstavi znake za konec vrste med vrstice izvorne datoteke.

2. Predprocesor vsako nagibnico, ki ji sledi nova vrsta, izbriše in tako efektivno stakne vrstice (§B.12.2).

3. Predprocesor razbije program na leksične elemente, ki jih ločujejo beli znaki; komentarje pri tem nadomesti z enim samim presledkom. Predprocesor nato upošteva svoje direktive in nadomesti makroje z nadomestnimi teksti (§§B.12.3-§B.12.10).
4. Predprocesor nadomesti ubežna zaporedja v znakovnih konstantah in nizih z njihovimi ekvivalenti; nato stakne sosednje nize.
5. Rezultat se nato prevede, poveže skupaj z drugimi programi in knjižnicami, s tem da se zbere potrebne programe in podatke in poveže eksterne funkcije in reference na objekte z njihovimi definicijami.

B.12.1 Trigrafična zaporedja

Množica znakov, ki jih `C` izvorni programi uporabljajo, je podmnožica sedembitnega `ASCII` koda, a pri tem vsebuje `ISO 646-1983 Invariant Code Set` kot podmnožico. Da bi lahko pisali programe v tej zmanjšani množici znakov, predprocesor zamenja vse primerke naslednjih trigrafičnih zaporedij z ustreznim znakom. Ta zamenjava se zgodi pred vsemi drugimi predelavami.

`??=`	`#`	`??(`	`[`	`??<`	`{`
`??/`	`\`	`??)`	`]`	`??>`	`}`
`??'`	`^`	`??!`	`|`	`??-`	`~`

Drugih zamenjav ni.

B.12.2 Lepljenje vrstic

Vrstice, ki se končujejo z znakom \, stakne predprocesor v eno vrstico tako, da zbriše nagibnico in sledeči znak za novo vrsto. To se zgodi preden predprocesor razbije tekst programa na leksične elemente.

B.12.3 Definicije makrojev in njihovo nadomeščanje

Kontrolna vrstica oblike

```
# define ime zaporedje-leksičnih-elementov
```

povzroči, da predprocesor zamenja vsak sledeči primerek imena z danim zaporedjem leksičnih elementov; beli znaki pred in za zaporedjem leksičnih elementov izginejo. Drug `#define` za isto ime je napačen, razen če je drugo zaporedje leksičnih elementov identično prvemu, potem ko so vsa podzaporedja belih znakov proglašena kot ekvivalentna.

Vrstica oblike

```
# define ime( seznam-imen ) zaporedje-leksičnih-elementov
```

kjer med prvim imenom in (ni nobenega presledka, je definicija makroja s parametri, podanimi s seznamom imen. Kot pri prvi obliki beli znaki pred in za zaporedjem leksičnih elementov izginejo in makro lahko ponovno definiramo samo, če ima nova definicija identično število identično črkovanih parametrov in identično zaporedje leksičnih elementov.

Kontrolna vrstica oblike

```
# undef ime
```

ukaže predprocesorju, naj pozabi definicijo imena ime. Direktivo `#undef` lahko uporabimo na neznanem imenu.

Če je bil makro definiran v drugi obliki, tekstualno sledeči primerki imena makroja, ki mu sledi morda bel prostor, okrogli predklepaj, zaporedje leksičnih elementov ločenih z vejico in nato okrogli zaklepaj, tvori klic makroja. Argumenti klica so zaporedja leksičnih elementov, ločenih z vejico; citirane vejice, ali pa vejice, ki so zaščitene z gnezdenimi oklepaji, no ločujejo argumentov. Med zbiranjem argumentov predprocesor ne zamenjuje makrojev. Število argumentov v klicu se mora ujemati s številom parametrov v definiciji. Potem ko so argumenti zbrani, predprocesor odstrani bel prostor okrog njih. Potem predprocesor zamenja vsako necitirano ime parametra z ustreznim argumentom. Razen v primeru, da se nadomestni tekst začenja z `#` ali začenja ali končuje z `##`, predprocesor pregleda leksične elemente, ki tvorijo argument, če ni morda treba zamenjati še kakšen makro v njem, in to tik preden nadomestni tekst zamenja ime makroja.

Dva specialna operatorja vplivata na proces zamenjave. Prvič, če pred primerkom parametra v nadomestnem tekstu stoji `#`, bo predprocesor okrog ustreznega parametra vstavil narekovaje `"` in potem `#` in ime parametra nadomestil s citiranim argumentom. Pred vsak znak `"` ali `\`, ki se pojavi v kakšnem nizu ali znakovni konstanti v argumentu, bo predprocesor vstavil znak `\`.

Drugič, če nadomestni tekst katerekoli vrste makroja vsebuje operator `##`, tedaj potem, ko so parametri zamenjani, predprocesor zbriše `##` skupaj z belim prostorom okrog njega, tako da se sosednji leksični elementi staknejo in tvorijo nov leksični element. Učinek je nedefiniran, če kot rezultat dobimo nekorekten leksični element ali pa če je rezultat odvisen od vrstnega reda procesiranja `##` operatorjev. Operator `##` ne sme biti na začetku ali na koncu nadomestnega teksta.

V obeh vrstah makrojev predprocesor ponovno pregleda zamenjan tekst, če ne vsebuje še kakšnega definiranega imena. Če je neko ime že bilo enkrat zamenjano, ga predprocesor ne bo ponovno zamenjal, če se ponovno pojavi med pregledovanjem; tako ime ostane nedotaknjeno.

Celo če se končna verzija zamenjanega teksta začenja z `#`, je predprocesor ne bo razumel kot svoje direktive.

Predprocesor lahko uporabljamo za manifestne konstante, na primer

```
#define TABSIZE 100
int table[TABSIZE];
```

Definicija

```
#define ABSDIFF(a, b)   ((a) > (b) ? (a) - (b) : (b) - (a))
```

definira makro, ki vrne absolutno vrednost razlike njegovih argumentov. Za razliko od funkcije, ki naj opravlja isto nalogo, so argumenti in vrnjena vrednost lahko poljubnega aritmetičnega tipa, lahko so celo kazalci. Če imata argumenta stranske učinke, se moramo zavedati, da bosta argumenta izračunana dvakrat, enkrat za potrebe testa in drugič za potrebe rezultata.

Če imamo definicijo

```
#define tempfile(dir)   #dir "/%s"
```

bo klic makroja `tempfile(/usr/tmp)` sproduciral

```
"/usr/tmp" "/%s"
```

kar bo predprocesor v nadaljevanju staknil v en sam niz. Po definiciji

```
#define cat(x, y)   x ## y
```

bo klic `cat(var, 123)` vrnil `var123`. Žal je klic `cat(cat(1, 2), 3)` nedefiniran: prisotnost `##` v zunanjem klicu prepreči, da bi predprocesor zamenjal argumenta. Zato dobimo zaporedje leksičnih elementov

```
cat ( 1 , 2 )3
```

in `)3` (stik zadnjega leksičnega elementa prvega argumenta s prvim leksičnim elementom drugega argumenta) ni legalen leksični element. Če vpeljemo posreden makro

```
#define xcat(x, y)  cat(x, y)
```

reči tečejo bolj gladko: `xcat(xcat(1, 2), 3)` vrne `123`, ker ekspanzija `xcat` ne vsebuje `##` operatorja.

Podobno, `ABSDIFF(ABSDIFF(a, b), c)` sproducira pričakovan v polnosti ekspandiran rezultat.

B.12.4 Vključevanje datotek

Kontrolna vrstica oblike

```
# include <filename>
```

povzroči zamenjavo te vrstice s celo vsebino datoteke `filename`. Znaki, ki tvorijo ime `filename`, ne smejo vsebovati `>` ali nove vrste, učinek pa je nepredvidljiv, če vsebujejo `"`, `'`, `\` ali `/*`. Imenovano datoteko predprocesor išče na zaporedju krajev, določenim z izvedbo.

Podobno, vrstica oblike

```
# include "filename"
```

išče najprej na mestu, povezanim z izvorno datoteko (fraza, ki je namenoma odvisna od izvedbe), nato pa še, če je bilo iskanje neuspešno, kot v prvi obliki. Učinek znakov `'`, `\` ali `/*` je spet nepredvidljiv, znak `>` pa je dovoljen.

Končno, direktiva oblike

```
# include zaporedje-leksičnih-elemnetov
```

ki se ne ujema z eno od gornjih oblik, je interpretirana tako, da se zaporedje leksičnih elementov ekspandira kot normalen tekst; rezultat mora biti ena od oblik `<...>` ali `"..."` in je potem obravnavan kot zgoraj.

Datoteke, navedene v `#include` direktivah, so lahko gnezdene.

B.12.5 Pogojno prevajanje

Dele programa lahko prevajamo pogojno, v skladu z naslednjo shematično sintakso.

predprocesorjev-pogoj:
  if-vrstica tekst elif-deli else-del$_{opt}$ `#endif`

if-vrstica:
  `# if` *konstantni-izraz*
  `# ifdef` *ime*
  `# ifndef` *ime*

elif-deli:
  elif-vrstica text
  elif-deli$_{opt}$

elif-vrstica:
  `# elif` *konstantni-izraz*

else-del:
  else-vrstica tekst

else-vrstica:
  `# else`

Vsaka od direktiv (*if-vrstica*, *elif-vrstica*, *else-vrstica* in `#endif`) stoji v vrstici sama zase. Konstantni izrazi v `#if` in sledečih `#elif` vrsticah se izračunavajo po vrsti dokler eden od njih nima neničelne vrednosti; tekst, ki sledi kontrolni vrstici z vrednostjo nič, predprocesor zavrže. Tekst, ki sledi kontrolni vrstici z neničelno vrednostjo, se normalno upošteva. "Tekst" se tu nanaša na vsak material, vključno predpocesorjeve vrstice, ki ni del pogojne strukture; lahko je prazen. Ko predprocesor enkrat najde uspešno `#if` ali `#elif` vrstico, in procesira njen tekst, predprocesor zavrže vse `#elif` in `#else` vrstice skupaj z njihovim tekstom. Če so vrednosti vseh izrazov nič in `#else` vrstica obstaja, je njen tekst obravnavan normalno. Tekst, ki ga kontrolirajo neaktivne veje pogoja, se zavrže z izjemo preverjanja vloženih pogojev.

Konstantni izrazi v `#if` in `#elif` konstruktih so podvrženi običajni makro substituciji. Dalje, vsak izraz oblike

```
defined ime
```

ali

```
defined ( ime )
```

predprocesor pred iskanjem imen makrojev zamenja z `1L`, če je ime predprocesorju definirano in z `0L`, če ni. Vsa imena, ki še ostanejo po ekspanziji makrojev, predprocesor zamenja z `0L`. Končno, vsaka celoštevična konstanta dobi pripono `L`, tako da je vsa aritmetika tipa `long` ali `unsigned long`.

Rezultirajoči konstantni izraz (§B.7.19) je omejen: biti mora celoštevilčen in ne sme vsebovati `sizeof`, prisil ali enumeracijskih konstant.

Kontrolni vrstici

```
#ifdef ime
#ifndef ime
```

sta ekvivalentni izrazoma

```
# if defined ime
# if ! defined ime
```

B.12.6 Kontrola oštevilčenja

Za potrebe drugih predprocesorjev, ki generirajo `C` programe, vrstica ene od oblik

```
# line konstanta "filename"
# line konstanta
```

prepriča prevajalnik, da nosi, za potrebe diagnostike napak, izvorna vrstica številko, podano z decimalno konstanto in da vrstica prihaja z datoteke imenovane `filename`. Če citirano ime datoteke manjka, se ime, ki ga vodi predprocesor, ne spremeni. Makroji v vrstici se ekspandirajo preden predprocesor vrstico interpretira.

B.12.7 Generiranje napak

Predprocesorjeva vrstica oblike

`# error zaporedje-leksičnih-elementov`$_{opt}$

povzroči, da predprocesor zapiše diagnostično sporočilo, ki vsebuje "zaporedje leksičnih elementov".

B.12.8 Pragmatični komentarji

Kontrolna vrstica oblike

`# pragma zaporedje-leksičnih-elementov`$_{opt}$

povzroči, da predprocesor podvzame dejanja, ki so odvisna od izvedbe. Neprepoznan pragmatičen komentar predprocesor zavrže.

B.12.9 Prazna direktiva

Direktiva oblike

```
#
```

je brez učinka.

B.12.10 Vnaprej definirana imena

Predprocesor pozna nekaj vnaprej definiranih imen, ki se ekspandirajo v posebno informacijo. Teh imen, tako kot predprocesorjevega operatorja `defined`, ne moremo oddefinirati ali ponovno definirati.

`__LINE__`	Decimalna konstanta, ki vsebuje trenutno številko izvorne vrstice.
`__FILE__`	Niz, ki vsebuje ime datoteke, ki se trenutno prevaja.
`__DATE__`	Niz, ki vsebuje datum prevajanja, v obliki `Mmm dd yyyy`.
`__TIME__`	Niz, ki vsebuje čas prevajanja, v obliki `hh:mm:ss`.
`__STDC__`	Konstanta `1`. Namen je, da je ta konstanta `1` samo v izvedbah, ki se pokoravajo `ANSI` standardu.

B.13 Povzetek slovnice

V §B.13.1 je rekapitulacija slovnice, ki smo jo uporabljali v začetnem delu tega dodatka. Ima natančno isto vsebino a v drugačnem vrstem redu in z imeni terminalnih simbolov v angleščini. V §B.13.2 navajamo angleško-slovenski slovar terminalnih simbolov.

B.13.1 Slovnica

V slovnici nastopajo nedefinirani terminalni simboli *integer-constant*, *character-constant*, *floating-constant*, *identifier*, *string* in *enumeration-constant*; besede in simbole v `Courier fontu` uporabljamo dobesedno. To slovnico lahko avtomatično pretvorimo v obliko, ki jo sintaktični analizatorji razumejo. Poleg tega, da dodamo sintaktične znake, ki so potrebni, da naznačijo alternative produkcij, moramo razdelati "one of" konstrukcije in (odvisno od pravil sintaktičnega analizatorja) podvojiti vsako produkcijo, ki se konča z $_{opt}$, enkrat z $_{opt}$ simbolom in drugič brez njega. Z eno dodatno spremembo, da zbrišemo produkcijo *typedef-name: identifier* in napravimo *typedef-name* za terminalni simbol, je taka slovnica sprejemljiva za sintaktični analizator `yacc`. Slovnica ima en konflikt, ki ga generira `if-else` dvoumje.

translation-unit:
  external-declaration
  translation-unit external-declaration

external-declaration:
  function-definition
  declaration

function-definition:
  declaration-specifiers$_{opt}$ *declarator declaration-list*$_{opt}$
  compound-statement

declaration:
  declaration-specifiers init-declarator-list$_{opt}$;

declaration-list:
  declaration
  declaration-list declaration

declaration-specifiers:
  storage-class-specifier declaration-specifiers$_{opt}$
  type-specifier declaration-specifiers$_{opt}$
  type-qualifier declaration-specifiers$_{opt}$

storage-class-specifier:
`auto`
`register`
`static`
`extern`
`typedef`

type-specifier:
`void`
`char`
`short`
`int`
`long`
`float`
`double`
`signed`
`unsigned`
struct-or-union-specifier
enum-specifier
typedef-name

type-qualifier:
`const`
`volatile`

struct-or-union-specifier:
struct-or-union identifier$_{opt}$ `{`*struct-declaration-list* `}`
struct-or-union identifier

struct-or-union:
`struct`
`union`

struct-declaration-list:
struct-declaration
struct-declaration-list struct-declaration

init-declarator-list:
init-declarator
init-declarator-list `,` *init-declarator*

init-declarator:
declarator
declarator `=` *initializer*

struct-declaration:
specifier-qualifier-list struct-declaration-list `;`

specifier-qualifier-list:
type-specifier specifier-qualifier-list$_{opt}$
type-qualifier specifier-qualifier-list$_{opt}$

struct-declaration-list:
struct-declarator
struct-declaration-list `,` *struct-declarator*

struct-declarator:
declarator
declarator$_{opt}$: *constant-expression*

enum-specifier:
enum identifier$_{opt}$ {*enumerator-list* }
enum identifier

enumerator-list:
enumerator
enumerator-list , *enumerator*

enumerator:
identifier
identifier = *constant-expression*

declarator:
pointer$_{opt}$ *direct-declarator*

direct-declarator:
identifier
(*declarator*)
direct-declarator [*constant-expression*$_{opt}$]
direct-declarator (*parameter-type-list*)
direct-declarator (*identifier-list*$_{opt}$)

pointer:
* *type-qualifier-list*$_{opt}$
* *type-qualifier-list*$_{opt}$ *pointer*

type-qualifier-list:
type-qualifier
type-qualifier-list type-qualifier

parameter-type-list:
parameter-list
parameter-list , ...

parameter-list:
parameter-declaration
parameter-list , *parameter-declaration*

parameter-declaration:
declaration-specifiers declarator
declaration-specifiers abstract-declarator$_{opt}$

identifier-list:
identifier
identifier-list , *identifier*

initializer:
assignment-expression
{*initializer-list* }
{*initializer-list* , }

initializer-list:
initializer
initializer-list `,` *initializer*

type-name:
specifier-qualifier-list abstract-declarator$_{opt}$

abstract-declarator:
pointer
pointer$_{opt}$ *direct-abstract-declarator*

direct-abstract-declarator:
`(` *abstract-declarator* `)`
direct-abstract-declarator$_{opt}$ `[` *constant-expression*$_{opt}$ `]`
direct-abstract-declarator$_{opt}$ `(` *parameter-type-list*$_{opt}$ `)`

typedef-name:
identifier

statement:
labeled-statement
expression-statement
compound-statement
selection-statement
iteration-statement
jump-statement

labeled-statement:
identifier `:` *statement*
`case` *constant-expression* `:` *statement*
`default` `:` *statement*

expression-statement:
expression$_{opt}$ `;`

compound-statement:
`{`*declaration-list*$_{opt}$ *statement-list*$_{opt}$ `}`

statement-list:
statement
statement-list statement

selection-statement:
`if` `(` *expression* `)` *statement*
`if` `(` *expression* `)` *statement* `else` *statement*
`switch` `(` *expression* `)` *statement*

iteration-statement:
`while` `(` *expression* `)` *statement*
`do statement while` `(` *expression* `)` `;`
`for` `(` *expression*$_{opt}$ `;` *expression*$_{opt}$ `;` *expression*$_{opt}$ `)` *statement*

jump-statement:
`goto` *identifier* `;`
`continue` `;`
`break` `;`
`return` *expression*$_{opt}$ `;`

expression:
assignment-expression
expression , *assignment-expression*

assignment-expression:
conditional-expresssion
unary-expression assignment-operator assignment-expression

assignment-operator: one of
= *= /= %= += -= <<= >>= &= ^= |=

conditional-expresssion:
logical-OR-expression
logical-OR-expression ? *expression* : *conditional-expresssion*

constant-expression:
conditional-expresssion

logical-OR-expression:
logical-AND-expression
logical-OR-expression || *logical-AND-expression*

logical-AND-expression:
inclusive-OR-expression
logical-AND-expression && *inclusive-OR-expression*

inclusive-OR-expression:
exclusive-OR-expression
inclusive-OR-expression | *exclusive-OR-expression*

exclusive-OR-expression:
AND-expression
exclusive-OR-expression ^ *AND-expression*

AND-expression:
equality-expression
AND-expression & *equality-expression*

equality-expression:
relational-expression
equality-expression == *relational-expression*
equality-expression != *relational-expression*

relational-expression:
shift-expression
relational-expression < *shift-expression*
relational-expression > *shift-expression*
relational-expression <= *shift-expression*
relational-expression >= *shift-expression*

shift-expression:
additive-expression
shift-expression << *additive-expression*
shift-expression >> *additive-expression*

additive-expression:
multiplicative-expression
additive-expression `+` *multiplicative-expression*
additive-expression `-` *multiplicative-expression*

multiplicative-expression:
cast-expression
multiplicative-expression `*` *cast-expression*
multiplicative-expression `/` *cast-expression*
multiplicative-expression `%` *cast-expression*

cast-expression:
unary-expression
`(` *type-name* `)` *cast-expression*

unary-expression:
postfix-expression
`++` *unary-expression*
`--` *unary-expression*
unary-operator cast-expression
`sizeof` *unary-expression*
`sizeof` `(` *type-name* `)`

unary-operator: one of
`&` `*` `+` `-` `~` `!`

postfix-expression:
primary-expression
postfix-expression `[` *expression* `]`
postfix-expression `(` *argument-expression-list*$_{opt}$ `)`
postfix-expression `.` *identifier*
postfix-expression `->` *identifier*
postfix-expression `++`
postfix-expression `--`

primary-expression:
identifier
constant
string
`(` *expression* `)`

argument-expression-list:
argument-expression
argument-expression-list `,` *argument-expression*

constant:
integer-constant
character-constant
floating-constant
enumeration-constant

Slovnica v nadaljevanju povzema predprocesorjevo slovnico kontrolnih vrstic, a ni primerna za mehanizirano prepoznavanje. Vključuje simbol *text*, ki pomeni običajen programski tekst, nepogojne predprocesorjeve kontrolne vrstice ali pa popolne pogojne konstrukte.

control-line:
`# define` *identifier token-sequence*
`# define` *identifier* (*identifier* , ... *identifier*) *token-sequence*
`# undef` *identifier*
`# include <` *filename* `>`
`# include "` *filename* `"`
`# include` *token-sequence*
`# line` *constant* " *filename* "
`# line` *constant*
`# error` *token-sequence*$_{opt}$
`# pragma` *token-sequence*$_{opt}$
`#`
preprocessor-conditional

preprocessor-conditional:
if-line text elif-parts else-part$_{opt}$ `# endif`

if-line:
`# if` *constant-expression*
`# ifdef` *identifier*
`# ifndef` *identifier*

elif-parts:
elif-line text
elif-parts$_{opt}$

elif-line:
`# elif` *constant-expression*

else-part:
else-line text

else-line:
`# else`

B.13.2 Angleško-slovenski slovar terminalnih simbolov

AND-expression	IN-izraz
additive-expression	aditivni-izraz
argument-expression	argumentni-izraz
argument-expression-list	seznam-argumentnih-izrazov
assignment-expression	prirejanje
assignment-operator	operator-prirejanja
cast-expression	prisiljeni-izraz
character-constant	znakovna-konstanta
compound-statement	sestavljeni-stavek
compound-statement	sestavljeni-stavek
conditional-expresssion	pogojni-izraz
constant-expression	konstantni-izraz
declaration	deklaracija
declaration-list	seznam-deklaracij
declaration-specifiers	določilo-deklaracije
enum-specifier	določilo-enumeracije

enumeration-constant	enumeracijska-konstanta
enumerator-list	seznam-enumeratorjev
equality-expression	enakostni-izraz
exclusive-OR-expression	ekskluzivni-ALI-izraz
expression	izraz
expression-statement	izrazni-stavek
external-declaration	eksterna-deklaracija
floating-constant	realna-konstanta
function-definition	definicija-funkcije
identifier	ime
inclusive-OR-expression	inkluzivni-ALI-izraz
init-declarator	init-deklarator
init-declarator-list	seznam-init-deklaratorjev
init-declarator-list	seznam-init-deklaratorjev
initializer	inicializator
integer-constant	celoštevilčna-konstanta
iteration-statement	iteracijski-stavek
jump-statement	skočni-stavek
labeled-statement	označeni-stavek
logical-AND-expression	loginčni-IN-izraz
logical-OR-expression	loginčni-ALI-izraz
multiplicative-expression	multiplikativni-izraz
one of	eden od
parameter-declaration	deklaracija-parametra
parameter-list	seznam-parametrov
parameter-type-list	seznam-tipov-parametrov
pointer	kazalec
postfix-expression	priponski-izraz
primary-expression	primarni-izraz
relational-expression	relacijski-izraz
selection-statement	izbirni-stavek
shift-expression	pomični-izraz
specifier-qualifier-list	spec-in-kval-seznam
statement	stavek
statement-list	seznam-stavkov
storage-class-specifier	določilo-pomnilniškega-razreda
string	niz
struct-declaration-list	seznam-struct-deklaracij
struct-declarator-list	seznam-struct-deklaracij
struct-or-union	struct-ali-union
struct-or-union-specifier	določilo-strukture-ali-unije
translation-unit	prevajalna-enota
type-qualifier	kvalifikator-tipa
type-qualifier-list	seznam-kvalifikatorjev-tipa
type-specifier	določilo-tipa
typedef-name	typedef-ime
unary-expression	unarni-izraz
unary-operator	unarni-operator

Dodatek C

Standardna knjižnica

Ta dodatek je povzetek knjižnice, definirane z ANSI standardom. Standardna knjižnica ni del pravega `C` jezika, a vsako okolje, ki podpira standardni `C`, nudi deklaracije funkcij, tipov in makro definicij te knjižnice. Opustili smo nekaj funkcij, ki so omejene uporabnosti in jih zlahka sestavimo iz preostalih; opustili smo široke znake in široke nize; in opustili smo diskusijo lokalov, to je lastnosti, ki so odvisne od lokalnega jezika, nacionalnosti ali kulture.

Funkcije, tipi in makroji standardne knjižnice so deklarirani v standardnih glavah:

```
<assert.h>  <float.h>   <math.h>    <stdard.h>  <stdlib.h>
<ctype.h>   <limits.h>  <setjmp.h>  <stddef.h>  <string.h>
<errno.h>   <locale.h>  <signal.h>  <stdio.h>   <time.h>
```

Do standardne glave pridemo z ukazom

```
#include <glava>
```

Standardne glave lahko vključujemo v poljubnem vrstnem redu in poljubno mnogokrat. Standardno glavo je treba vključiti zunaj eksternih deklaracij ali definicij in preden uporabimo karkoli iz nje. Standardna glava ni treba da je izvorna datoteka.

Eksterna imena, ki se začenjajo s podčrtajem, so rezervirana za knjižnico, tako kot vsa druga imena, ki se začenjajo s podčrtajem in veliko črko ali še enim podčrtajem.

C.1 Vhod in izhod: stdio.h

Funkcije za branje in pisanje, tipi in makroji, definirani v `<stdio.h>`, predstavljajo skoraj tretjino knjižnice.

Tok je izvor ali ponor podatkov, povezan z diskom ali kakšno drugo periferno enoto. Knjižnica podpira tekstovne in binarne tokove, čeprav so ti na nekaterih sistemih[1] identični. Tekstovni tok je zaporedje vrstic; vsaka vrstica vsebuje nič ali več znakov in se konča z znakom `\n`. Okolje lahko zahteva prevajanje med tekstovnim tokom in kakšno drugačno reprezentacijo (, na primer preslikavanje `\n` v pomik na začetek vrste in pomik v novo vrstico). Binarni tok je zaporedje neprocesiranih zlogov, ki predstavljajo interne podatke, z lastnostjo, če tok zapišemo in nazaj preberemo na istem sistemu, dobimo iste zloge.

---

[1] na primer `UNIX`

Tok povežemo z datoteko ali z napravo tako, da ga odpremo; ta povezava se prekine, ko tok zapremo. Odpiranje datoteke vrne kazalec na objekt tipa `FILE`, ki hrani vso potrebno informacijo za kontrolo toka. Uporabljali bomo izraza "kazalec na datoteko" in "tok" kot sinonima, če ne bo možnosti, da bi se kaj zamešalo.

Ko program začne z izvajanjem, so trije tokovi, `stdin`, `stdout` in `stderr` že odprti.

C.1.1 Operacije na datotekah

Naslednje funkcije imajo opravka z operacijami na datotekah. Tip `size_t` je nepredznačen celoštevilčen tip, ki ga vrne operator `sizeof`.

`FILE *fopen(const char *filename, const char *mode)`

`fopen` odpre imenovano datoteko in vrne tok ali `NULL`, če se poskus ponesreči.
Legalne vrednosti načina `mode` vključujejo

`"r"`	odpri tekstovno datoteko za branje
`"w"`	kreiraj tekstovno datoteko za pisanje; odvrzi prejšnjo vsebino, če obstaja
`"a"`	dodaj; kreiraj ali odpri tekstovno datoteko za pisanje na koncu datoteke
`"r+"`	odpri tekstovno datoteko za ažuriranje (to je, za branje in pisanje)
`"w+"`	kreiraj tekstovno datoteko za ažuriranje; odvrzi prejšno vsebino, če obstaja
`"a+"`	dodaj; odpri ali kreiraj tekstovno datoteko za ažuriranje, pisanje na koncu

Način ažuriranje dovoljuje branje in pisanje iste datoteke; med branjem in pisanjem ali obratno moramo poklicati funkcijo `fflush` ali kakšno funkcijo za pozicioniranje datoteke. Če način `mode` vsebuje `b` poleg začetne črke, kot v `"rb"` ali `"w+b"`, to označuje binarno datoteko. Imena datotek so omejena na `FILENAME_MAX` znakov. Največ `FOPEN_MAX` datotek je lahko hkrati odprtih.

`FILE *freopen(const char *filename, const char *mode,`
`* FILE *stream)`

Funkcija `freopen` odpre datoteko na predpisan način `mode` in ji pridruži tok `stream`. Funkcija vrne `stream` ali `NULL` v primeru napake. Funkcijo `freopen` običajno uporabljamo, da spremenimo datoteke, povezane z `stdin`, `stdout` in `stderr`.

`int fflush(FILE *stream):`

Na izhodnem toku `fflush` sproži izpis vseh neizpisanih podatkov, ki čakajo v izravnalnikih; na vhodnem toku je učinek nedefiniran. Funkcija vrne `EOF` v primeru napake pri pisanju, sicer pa ničlo.

`int fclose(FILE *stream)`
`fclose` izpiše vse neizpisane podatke na izhodnem toku, odvrže vse neprebrane podatke v vhodnih izravnalnikih, sprosti vse avtomatično dodeljene izravnalnike in zapre tok `stream`. Funkcija vrne `EOF` v primeru napake, sicer pa ničlo.

`int remove(const char *filename)`
`remove` odstrani imenovano datoteko tako, da bodo sledeči poskusi odpiranja spodleteli. Funkcija vrne neničelno vrednost, če poskus odstranjevanja spodleti.

`int rename(const char *oldname, const char *newname)`
`rename` spremeni ime datoteke; funkcija vrne neničelno vrednost, če poskus spodleti.

`FILE *tmpfile(void)`
`tmpfile` kreira začasno datoteko na način `"wb+"`; ta datoteka bo avtomatično odstranjena, ko jo bo program zaprl ali ko bo program normalno končal z delom. Funkcija `tmpfile` vrne tok ali `NULL`, če datoteke ni mogoče kreirati.

`char *tmpnam(char s[L_tmpnam])`
`tmpnam(NULL)` kreira niz, ki ni ime obstoječe datoteke in vrne kazalec na interni statični večkratni znak. Klic `tmpnam(s)` shrani niz v večkraten znak `s` poleg tega, da ga vrne kot funkcijsko vrednost; `s` mora imeti prostora za vsaj `L_tmpnam` znakov. Funkcija `tmpnam` generira vsakokrat, ko jo pokličemo, drugačno ime; funkcija `tmpnam` garantira vsaj `TMP_MAX` različnih imen med izvajanjem programa. Pazite, `tmpnam` kreira ime, ne datoteke.

`int setvbuf(FILE *stream, char *buf, int mode, size_t size)`
`setvbuf` kontrolira izravnalnike toka; poklicati jo moramo pred branjem in pisanjem. Z `mode` `_IOFBF` dobimo polno izravnavanje, z `_IOLBF` izravnavanje vrstic na tekstovnih datotekah in z `_IONBF` izravnavanja ni. Če je `buf` različen od `NULL`, bo to izravnalnik; v nasprotnem primeru bo datoteki izravnalnik dodeljen. Vrednost `size` določi velikost izravnalnika. Funkcija `setvbuf` vrne pri vsaki napaki od nič različno vrednost.

`void setbuf(FILE *stream, char *buf)`
Če je `buf` `NULL`, se izravnavanje datoteke `stream` izključi. V nasprotenm primeru je setbuf ekvivalenten `(void)setvbuf(stream, buf, _IOFRBF, BUFSIZ)`.

C.1.2 Formatiran izpis

Funkcija `fprintf` nudi pretvarjanje formatiranega izhoda.

```
int fprintf(FILE *stream, const char *format, ...)
```

Funkcija `fprintf` prevede in zapiše izhod na `stream` pod kontrolo formata. Funkcija vrne število izpisanih znakov ali negativno število, če je prišlo do napake.

Formatni niz vsebuje objekte dve tipov: običajne znake, ki jih `fprintf` enostavno prepiše na izhodni tok in pretvorna določila, od katerih vsako povzroči pretvarjanje in izpis naslednjega argumenta funkcije `fprintf`. Vsako pretvorno določilo se začenja z znakom `%` in konča s pretvornim znakom. Med znakom `%` in pretvornim znakom so lahko, v tem vrstnem redu

- Modifikatorji (v poljubnem vrstem redu), ki modificirajo določilo:
  - Znak minus, ki predpisuje levo poravnavanje prevedenega argumenta.
  - Znak plus, ki predpisuje, da mora biti število vedno izpisano s predznakom.
  - Presledek, ki pravi, da se mora zapisano število začeti s presledkom ali z minusom.
  - Ničla, ki določa za numeričen argument popolnitev z vodilnimi ničlami.
  - Znak `#`, ki pomeni alternativen format:
    - določilo `o`: prva števka bo ničla;
    - določilo `x`, `X`: pred neničelnim številom bo `0x` ali `0X`;
    - določilo `e`, `E`, `f`, `g`, `G`: število bo vedno imelo decimalno piko;
    - določilo `g`, `G`: ne odstrani ničel na koncu.
- Število, ki določa minimalno širino polja. Preveden argument bo zasedel vsaj toliko znakov. Če ima preveden argument manj kot širina polja znakov, bo popolnjen na levi (na desni, če je bilo zahtevano levo poravnavanje) do širine polja. Znak za popolnitev je običajno presledek, ničla, če smo to zahtevali z modifikatorjem.
- Pika, ki ločuje minimalno širino polja od natančnosti.
- Število, natančnost, ki v primeru niza določa maksimalno število znakov, v primeru `e`, `E` ali `f` pretvorbe število števk za decimalno piko, število pomembnih števk za `g` in `G` pretvarjanje, in v primeru celega števila minimalno število števk (po potrebi dodaj vodilne ničle).
- Črka `h`, če je število, namenjeno za izpis, tipa `short` ali `unsigned short`, črka `l`, če je število, namenjeno za izpis, tipa `long` ali `unsigned long`, črka `L`, če je število, namenjeno za izpis, tipa `long double`.

Širino polja in/ali natančnost lahko predpišemo kot `*`. V tem primeru bo funkcija `fprintf` vzela naslednji argument, ki mora biti tipa `int`, in ga uporabila kot širino polja ali natančnost.

Formatni znaki, ki ji `fprintf` razume, so prikazani v tabeli C.1.

`int printf(const char *format, ...)`

Funkcija `printf(...)` je ekvivalentna funkciji `fprintf(stdout, ...)`.

`int sprintf(char *string, const char *format, ...)`

Funkcija `sprintf` prevaja znake enako kot funkcija `printf`, le prevedene znake skupaj s končnim `\0` shrani v niz `string`. Niz `string` mora biti dovolj velik, da lahko shrani vse generirane znake. Število, ki ga funkcija vrne, ne vključuje `\0`.

```
int vprintf(const char *format, va_list arg)
int vfprintf(FILE *fp, const char *format, va_list arg)
int vsprintf(char *s, const char *format, va_list arg)
```

Funkcije `vprintf`, `vfprintf` in `vsprintf` so ekvivalentne ustreznim `printf` funkcijam, razen da je seznam variabilnih argumentov nadomeščen z `arg`, ki je bil inicializiran z `va_start` makrojem in morda še s klici `va_arg` makroja. Glej diskusijo `<stdarg.h>` v sekciji C.7

Tabela C.1: Formatni znaki funkcije fprintf

znak	tip argumenta	vrednost izpisana kot
`d`, `i`	`int`	Desetiško število.
`o`	`unsigned`	Nepredznačeno osmiško število (brez vodilne ničle).
`x`, `X`	`unsigned`	Nepredznačeno šestnajstiško število (brez vodilnega `0x` ali `0X`) z uporabo `abcdef` ali `ABCDEF` za števke `10`, ..., `15`.
`u`	`unsigned`	Nepredznačeno desetiško število.
`c`	`int`	En sam znak.
`s`	`char *`	Znaki iz niza dokler ne dosežemo konca niza ali ni izpisano dovolj znakov (natančnost).
`f`	`double`	`[-]m.dddddd`, kjer je število števk `d` določeno z natančnostjo. Natančnost nič zatre decimalno piko.
`e`, `E`	`double`	`[-]m.dddddd{eE}{+-}xx`, kjer je število števk `d` določeno z natančnostjo. Natančnost nič zatre decimalno piko.
`g`, `G`	`double`	Uporabi `e` ali `E` format, če je eksponent manjši od `-4` ali če je eksponent večji ali enak natančnosti, sicer pa uporabi typef format. Ničle in pike na koncu se ne pišejo.
`p`	`void *`	Kazalec (izpis je odvisen od izvedbe).
`n`	`int *`	Število znakov, ki je bilo izpisano doslej, se shrani v argument.
`%`	—	Izpis znaka `%`.

C.1.3 Formatirano branje

Funkcije `scanf` opravljajo pretvorbe formatiranega branja.

```
int fscanf(FILE *stream, const char *format, ...)
```

`fscanf` bere znake s toka `stream` in jih dekodira tako, kot določa formatni niz `format`. Rezultate prevajanja `fscanf` shranjuje v spremenljivke, ki so identificirane s preostalimi argumenti, ki *morajo* biti kazalci. Funkcija `fscanf` se vrne, ko je format izčrpan. Funkcija vrne `EOF`, če se je konec datoteke ali napaka pojavila, še preden je funkcija kaj pretvorila, drugače pa vrne število pretvorjenih in prirejenih postavk.

Formatni niz običajno vsebuje formatna določila, ki dirigirajo interpretacijo vhodnih znakov. Formatni niz lahko vsebuje:

- Bele znake, ki jih `fscanf` ignorira.
- Običajen znak (ne `%`), ki se mora ujemati z naslednjim nebelim znakom na vhodnem toku.
- Formatna določila, ki vsebujejo, v tem vrstem redu:
  - znak `%`,
  - neobvezen znak za prepoved prirejanja `*`,
  - neobvezno število, ki določa maksimalno širino polja,
  - neobvezen znak `h`, `l` ali `L`, ki določa širino tarče
  - in formatni znak.

Formatno določilo določa pretvarjanje naslednjega vhodnega polja. Normalno se rezultat shrani v spremenljivko, na katero kaže naslednji argument, ki mora biti kazalec. Če pretvobeno določilo vsebuje znak za prepoved prirejanja `*`, tedaj bo `fscanf` samo prebrala in preverila naslednje vhodno polje, priredila pa ga ne bo nikamor. V takem primeru tudi kazalca za shranjevanje ne potrebujemo. Vhodno polje je definirano kot zaporedje nebelih znakov in sega do naslednjega belega znaka ali pa dokler širina polja ni izčrpana. To pomeni, da bo funkcija `scanf` preskakovala znake za konec vrste, da bo našla svoj vhod. (Beli znaki so presledek, predelčnik, nova vrsta, povratek voza, vertikalni predelčnik in nova stran.)

Formatni znaki, ki jih `fscanf` razume, so prikazani v tabeli C.2.

Pred pretvornimi znaki `d`, `i`, `n`, `o`, `u` ali `x` lahko napišemo še `h`, ki pomeni, da je argument `short *` namesto `int *` ali pa `l`, ki pomeni, da je argument `long *` namesto `int *`. Podobno lahko pred pretvorbene znake `e`, `f` ali `g` napišemo `l`, ki pomeni, da je argument `double *` namesto `float *`, ali pa `L`, ki pomeni, da je argument `long double *` namesto `float *`.

`int scanf(const char *format, ...)`
Funkcija `scanf(...)` je identična z `fscanf(stdin, ...)`.

`int sscanf(const char *s, const char *format, ...)`
Funkcija `sscanf(s, ...)` je ekvivalentna funkciji `scanf(...)` razen, da vhodni znaki prihajajo iz niza `s` namesto s standardnega vhoda.

Tabela C.2: Formatni znaki funkcije `fscanf`

znak	tip argumenta	vhodni niz
`d`	`int *`	Desetiško celo število.
`i`	`int *`	Celo število, ki je lahko zapisano osmiško (vodilna ničla) ali šestnajstiško (vodilna 0x ali 0X).
`o`	`unsigned *`	Osmiško število (z ali brez vodilne ničle).
`u`	`unsigned *`	Nepredznačeno celo število.
`x`	`unsigned *`	Šestnajstiško število (z ali brez 0x ali 0X).
`c`	`char *`	Znaki. Vhodni znaki, ki so na vrsti, (privzeto eden), se shranijo na naznačeno mesto brez končnega praznega znaka. Običajen preskok belih znakov se ne zgodi. Če želite prebrati naslednji nebel znak, uporabite `%1s`.
`s`	`char *`	Niz znakov (brez narekovajev). Argument (kazalec) mora kazati na večkraten znak, v katerem je dovolj prostora za vse znake in za zaključni prazen znak.
`e`, `f`, `g`	`float *`	Realno število z neobveznim predznakom, neobvezno decimalno piko in neobveznim eksponentom.
`p`	`void *`	Kazalec, kot ga izpiše `printf("%p")`.
`n`	`int *`	Shrani v argument število znakov prebranih doslej. Ne preberi ničesar. Števec shranjenih vrednosti ostane nespremenjen.
`[...]`	`char *`	Preberi najdaljši neprazen niz znakov iz množice `[...]`. Prebrani niz je zaključen z `\0`. `[]...]` pomeni, da je `]` v množici.
`[^...]`	`char *`	Preberi najdaljši neprazen niz znakov, ki niso v množici `[...]`. Prebrani niz je zaključen z `\0`. `[^]...]` pomeni, da je `]` v množici.
`%`		Dobeseden znak `%` brez prirejanja.

C.1.4 Funkcije za branje in pisanje znakov

`int fgetc(FILE *stream)`
`fgetc` vrne naslednji znak s toka `stream` kot nepredznačen znak (prisiljen v `int`), ali `EOF`, če se pojavi konec datoteke ali kakšna napaka.

`char *fgets(char *s, int n, FILE *stream)`
`fgets` prebere kvečjemu naslednjih `n-1` znakov v večkratno vrednost; branja je konec, če naletimo na znak za novo vrsto; znak za novo vrsto se tudi shrani v večkraten znak, tako kot prazen znak na koncu niza. `fgets` vrne `s` ali `NULL`, če je naletela na konec datoteke ali napako.

`int fputc(int c, FILE *stream)`
`fputc` zapiše znak `c` (pretvorjen v `unsigned char`) na tok `stream`. Funkcija vrne zapisan znak ali `EOF` v primeru napake.

`int fputs(const char *s, FILE *stream)`
`fputs` zapiše niz `s` (, ki ni treba, da vsebuje znak za novo vrsto) na tok `stream`; funkcija vrne nenegativno vrednost ali `EOF` v primeru napake.

`int getc(FILE *stream)`
`getc` je ekvivalentna `fgetc` razen da je makro, ki lahko izračuna tok `stream` več kot enkrat.

`int getchar(void)`
`getchar` je ekvivalentna `getc(stdin)`.

`char *gets(char *s)`
`gets` prebere naslednjo vrstico v večkraten znak `s`; `gets` nadomesti končen znak za novo vrsto s praznim znakom. Funkcija vrne `s`, v primeru konca datoteke ali napake pa `NULL`.

`int putc(int c, FILE *stream)`
`putc` je ekvivalentna `fputc` razen da je makro, ki lahko izračuna tok `stream` več kot enkrat.

`int putchar(int c)`
`putchar(c)` je ekvivalentno `putc(c, stdout)`.

`int puts(const char *s)`
`puts` zapiše niz `s` in novo vrsto na `stdout`. Funkcija vrne nenegativno vrednost, v primeru napake pa `EOF`.

`int ungetc(int c, FILE *stream)`
`ungetc` vrne `c` (pretvorjen v `unsigned char`) na tok `stream`, kjer bo na voljo za naslednje branje. Funkcija garantira samo en vrnjen znak za vsak tok. Znaka `EOF` ne moremo vrniti. Funkcija `ungetc` vrne vrnjen znak, v primeru napake pa `EOF`.

C.1.5 Funkcije za direktno branje in pisanje

`size_t fread(void *ptr, size_t size, size_t nobj, FILE *stream)`
`fread` prebere s toka `stream` v večkratno vrednost `ptr` kvecjemu `nobj` objektov velikosti `size`. Funkcija `fread` vrne število prebranih objektov; to je morda manj od zahtevanega števila. Stanje določimo s funkcijama `feof` in `ferror`.

`size_t fwrite(const void *ptr, size_t size, size_t nobj,`
`FILE *stream)`

`fwrite` zapiše z večkratne vrednosti `ptr` `nobj` objektov velikosti `size` na tok `stream`. Funkcija vrne število zapisanih objektov, ki je v primeru napake manj kot `nobj`.

C.1.6 Funkcije za pozicioniranje datotek

`int fseek(FILE *stream, long offset, int origin)`

`fseek` pozicionira tok `stream`; naslednje branje ali pisanje bo dosegalo podatke začenši na novi poziciji. Za binarno datoteko lahko pozicijo postavimo `offset` zlogov od `origin`, ki je lahko `SEEK_SET` (začetek), `SEEK_CUR` (trenuten položaj) ali `SEEK_END` (konec datoteke). Za tekstovni tok mora biti `offset` enak nič ali pa vrednost, ki jo vrne `ftell` (in v tem primeru mora biti `origin` `SEEK_SET`). Funkcija `fseek` vrne od nič različno vrednost v primeru napake.

`long ftell(FILE *stream)`

`ftell` vrne trenutno pozicijo toka `stream` ali `-1L` v primeru napake.

`void rewind(FILE *stream)`

klic funkcije `rewind(fp)` je ekvivalenten klicu funkcije`fseek(fp, 0L, SEEK_SET); clearerr(fp)`.

`int fgetpos(FILE *stream, fpos_t *ptr)`

`fgetpos` zapiše trenutni položaj toka `stream` v `*ptr`, da ga bomo lahko uporabili v funkciji `fsetpos`. Tip `fpos_t` je primeren za beleženje takih vrednosti. Funkcija `fgetpos` vrne v primeru napake neničelno vrednost.

`int fsetpos(FILE *stream, const fpos_t *ptr)`

`fsetpos` pozicionira tok `stream` na pozicijo, ki jo je `fgetpos` zapisal v `*ptr`. Funkcija `fsetpos` vrne v primeru napake neničelno vrednost.

C.1.7 Funkcije za napake

Mnogo funkcij v knjižnici postavi statusni indikator, če se zgodi kakšna napaka ali če funkcija med branjem naleti na konec datoteke. Te indikatorje lahko postavljamo in testiramo eksplicitno. Poleg tega lahko celoštevilčni izraz `errno` (deklariran v `<errno.h>`) vsebuje številko napake, ki daje podrobnejšo informacijo a zadnji napaki.

`void clearerr(FILE *stream)`

`clearerr` zbriše indikatorja konca datoteke in napake na toku `stream`.

`int feof(FILE *stream)`

`feof` vrne neničelno vrednost, če je na toku `stream` postavljen indikator konca datoteke.

`int ferror(FILE *stream)`

`ferror` vrne neničelno vrednost, če je na toku `stream` postavljen indikator napake.

`void perror(const char *s)`

`perror(s)` natiska `s` in od izvedbe odvisno sporočilo o napaki, ki ustreza celemu številu v `errno`.

C.2 Testi znakovnih razredov: ctype.h

Glava `<ctype.h>` deklarira funkcije za testiranje znakov. V vsaki funkciji mora biti njen argument nek `int` z vrednostjo, ki mora biti `EOF` ali pa predstavljiva kot `unsigned char`. Vrednost funkcij je `int`. Funkcije vrnejo neničelno vrednost (true), če argument `c` zadošča opisanemu pogoju in nič, če ne.

`int isalnum(int c)`
`isalpha(c)` ali `isdigit(c)`

`int isalpha(int c)`
`isupper(c)` ali `islower(c)`

`int iscntrl(int c)`
kontrolni znak

`int isdigit(int c)`
desetiška števka

`int isgraph(int c)`
izpisljiv znak razen presledka

`int islower(int c)`
mala črka

`int isprint(int c)`
izpisljiv znak vključno s presledkom

`int ispunct(int c)`
izpisljiv znak razen presledka, črke ali števke

`int isspace(int c)`
presledek, nova stran, nova vrsta, povratek valja, predelčnik, vertikalni predelčnik

`int isupper(int c)`
velika črka

`int isxdigit(int c)`
šestnajstiška stevka

V sedembitnem `ASCII` kodu so izpisljivi znaki `0x20` (`' '`) do `0x7E` (`'~'`); kontrolni znaki so `0` (`NUL`) do `0x1F` (`US`) in `0x7F` (`DEL`).

Poleg tega obstajata še dve funkciji za pretvarjanje velikih črk v male in nazaj:

`int tolower(int c)`
pretvori `c` v malo črko

`int toupper(int c)`
pretvori `c` v veliko črko

Če je `c` velika črka, `tolower(c)` vrne ustrezno malo črko; sicer vrne `c`. Če je `c` mala črka, `toupper(c)` vrne ustrezno veliko črko; sicer vrne `c`.

C.3 Funkcije za manipulacije nizov: string.h

V glavi `<string.h>` obstajata dve skupini funkcij za manipulacije z nizi. Prva skupina ima imena, ki se začenjajo s `str`; druga skupina ima imena, ki se začenjajo z `mem`. Razen za `memmove` je obnašanje nedefinirano, če se objekta, ki ju zajema prepisovanje, prekrivata.

`char *strcpy(char *s, const char *ct)`
prepiši niz `ct`, vključno s praznim znakom, v niz `s`; vrni `s`.

`char *strncpy(char *s, const char *ct, size_t n)`
prepiši kvečjemu `n` znakov niza `ct` v niz `s`; vrni `s`. Dodaj prazen znak, če ima `ct` manj kot `n` znakov.

`char *strcat(char *s, const char *ct)`
prilepi niz `ct` na konec niza `s`; vrni `s`.

`char *strncat(char *s, const char *ct, size_t n)`
prilepi kvečjemu `n` znakov niza `ct` na konec niza `s`, zaključi niz `s` s praznim znakom; vrni `s`.

`int strcmp(const char *cs, const char *ct)`
primerjaj niz `cs` z nizom `ct`; vrni `<0`, če je `cs<ct`, `0`, če je `cs==ct` in `>0`, če je `cs>ct`.

`int strncmp(const char *cs, const char *ct, size_t n)`
primerjaj kvečjemu `n` znakov niza `cs` z nizom `ct`; vrni `<0`, če je `cs<ct`, `0`, če je `cs==ct` in `>0`, če je `cs>ct`.

`char *strchr(const char *cs, int c)`
vrni kazalec na prvi primerek znaka `c` v nizu `cs` ali `NULL`, če znaka `c` v nizu `cs` ni.

`char *strrchr(const char *cs, int c)`
vrni kazalec na zadnji primerek znaka `c` v nizu `cs` ali `NULL`, če znaka `c` v nizu `cs` ni.

`size_t strspn(const char *cs, const char *ct)`
vrni dolžino predpone niza `cs` tvorjene iz znakov v nizu `ct`.

`size_t strcspn(const char *cs, const char *ct)`
vrni dolžino predpone niza `cs` tvorjene iz znakov, ki niso v nizu `ct`.

`char *strpbrk(const char *cs, const char *ct)`
vrni kazalec na prvi primerek kateregakoli znaka iz niza `ct` v nizu `cs`, `NULL` če ga ni.

`char *strstr(const char *cs, const char *ct)`
vrni kazalec na prvi primerek niza `ct` v nizu `cs`, `NULL`, če ga ni.

`size_t strlen(const char *cs)`
vrni dolžino niza `cs`.

`char *strerror(int n)`
vrni kazalec na niz znakov, odvisen od izvedbe, ki ustreza napaki `n`.

`char *strtok(char *s, const char *ct)`
v nizu `s` poišči leksični element, omejen z znaki iz niza `ct`; glej spodaj.

Zaporedje klicev funkcije `strtok(s, ct)` razbije niz `s` na leksične elemente, ki so ločeni z znaki iz niza `ct`. Prvi klic v zaporedju klicev mora imeti argument `s` različen od `NULL` in poišče prvi leksični element, sestavljen iz znakov, ki niso v nizu `ct`; `strtok` konča leksični element s tem, da v nizu `s` prepiše naslednji znak z `\0`, in vrne kazalec na leksični element. Vsak naslednji klic, ki ima `NULL` za parameter `s`, vrne kazalec na naslednji leksični element s tem, da začenja iskanje takoj za prejsnjim leksičnim elementom. Funkcija `strtok` vrne `NULL`, če ne najde novega leksičnega elementa. Niz `ct` je lahko pri vsakem klicu drugačen.

Funkcije `mem...` so namenjene za manipulacije objektov kot da so ti objekti večkratni znaki; namen je dostop do učinkovitih funkcij.

`void *memcpy(void *s, const void *ct, size_t n)`
prepiši `n` zlogov iz `ct` v `s` in vrni `s`.

`void *memmove(void *s, const void *ct, size_t n)`
isto kot `memcpy`, le da funkcionira tudi, če se objekta prekrivata.

`int memcmp(const void *cs, const void *ct, size_t n)`
primerjaj prvih `n` zlogov objekta `cs` z objektom `ct`; vrnjena vrednost je kot pri `strcmp`.

`void *memchr(const void *cs, int c, size_t n)`
vrni kazalec na prvi primerek zloga `c` v objektu `cs`, `NULL`, če zloga `c` ni v objektu `cs` med prvimi `n` zlogi.

`void *memset(void *s, int c, size_t n)`
shrani zlog `c` v prvih `n` zlogov objekta `s`, vrni `s`.

C.4 Matematične funkcije: math.h

Glava `<math.h>` deklarira matematične funkcije in makroje.

Makroja `EDOM` in `ERANGE` (najdemo ju v `<errno.h>`) sta definirana kot od nič različni celoštevilčni konstanti in ju uporabljamo za sporočnje napak o domeni in zalogi vrednosti; `HUGE_VAL` je pozitivna vrednost tipa `double`. Napaka domene pomeni, da je argument zunaj domene, na kateri je funkcija definirana. V primeru napake domene `errno` dobi vrednost `EDOM`; vrnjena vrednost funkcije je prepuščena izvedbi. Napaka zaloge vrednosti pomeni, da vrednosti funkcije ne moremo predstaviti kot število tipa `double`. Če rezultat prekorači obseg, funkcija vrne `HUGE_VAL` s pravim predznakom in postavi `errno` na `ERANGE`. Če rezultat podkorači obseg, funkcija vrne ničlo; odločitev, ali funkcija postavi `errno` na `ERANGE`, je prepuščena izvedbi.

V nadaljevanju so naštete "matematične" funkcije.

`double sin(double x)`
sinus kota v radianih $\sin x$.

`double cos(double x)`
kosinus kota v radianih $\cos x$.

`double tan(double x)`
tangens kota v radianih $\tan x$.

`double asin(double x)`
$\arcsin x$ z vrednostjo na intervalu $[-\pi/2, \pi/2]$, $x \in [-1, 1]$.

`double acos(double x)`
$\arccos x$ z vrednostjo na intervalu $[0, \pi]$, $x \in [-1, 1]$.

`double atan(double x)`
$\arctan x$ z vrednostjo na intervalu $[-\pi/2, \pi/2]$.

`double atan2(double y, double x)`
polarni kot točke (x, y) na intervalu $(-\pi, \pi]$.

`double sinh(double x)`
hiperbolični sinus $\sinh x$.

`double cosh(double x)`
hiperbolični kosinus $\cosh x$.

`double tanh(double x)`
hiperbolični tangens $\tanh x$.

`double exp(double x)`
eksponentna funkcija $\exp x$.

`double log(double x)`
naravni logaritem $\ln x, x > 0$.

`double log10(double x)`
desetiški (Briggsov) logaritem $\log_{10} x, x > 0$.

`double pow(double x, double y)`
x^y. Če je $x = 0$ in $y <= 0$, ali če je $x < 0$ in y ni celo število, dobimo napako domene.

`double sqrt(double x)`
kvadratni koren $\sqrt{x}, x >= 0$.

`double ceil(double x)`
najmanjše celo število $>= x$, vrnjeno kot `double`.

`double floor(double x)`
največje celo število $<= x$, vrnjeno kot `double`.

`double fabs(double x)`
absolutna vrednost $|x|$.

`double ldexp(double x, int n)`
$x \times 2^n$

`double frexp(double x, int *exp)`
razdeli x v normaliziran ulomek na intervalu $[1/2, 1)$, ki ga funkcija vrne, in eksponent potence 2, ki se shrani v `*exp`. Če je $x = 0$, sta obe komponenti rezultata nič.

`double modf(double x, double *ip)`
razdeli x v celi in ulomljeni del, od katerih ima vsak isti predznak kot x. Funkcija shrani celi del v `*ip` in vrne ulomljeni del.

`double fmod(double x, double y)`
realen ostanek deljenja x/y z istim predznakom kot x. Če je $y = 0$, je rezultat odvisen od izvedbe.

C.5 Druge porabne funkcije: stdlib.h

Glava `<stdlib.h>` deklarira funkcije za pretvarjanje števil, dodeljevanje pomnilnika in podobne posle.

`double atof(const char *s)`
`atof` pretvori `s` v `double` vrednost; funkcija je ekvivalentna klicu `strtod(s, (char **)NULL)`.

`int atoi(const char *s)`

`atoi` pretvori niz `s` v celo število; funkcija je ekvivalentna klicu `(int)strtol(s, (char **)NULL, 10)`.

`long atol(const char *s)`

`atol` pretvori niz `s` v `long` število; funkcija je ekvivalentna klicu `strtol(s, (char **)NULL, 10)`.

`double strtod(const char *s, char **endp)`

`stdtod` pretvori glavo `s` v `double` in pri tem ignorira vodilne bele znake; funkcija shrani kazalec na neprocesiran rep v `*endp` razen če je `endp` enak `NULL`. Če bi odgovor prekoračil obseg, funkcija vrne `HUGE_VAL` s pravilnim predznakom; če bi odgovor podkoračil obseg, funkcija vrne nič. V obeh primerih funkcija postavi `errno` na `ERANGE`.

`long strtol(const char *s, char **endp, int base)`

`strtol` pretvori glavo `s` v `long` in pri tem ignorira vodilne bele znake; funkcija shrani kazalec na neprocesiran rep v `*endp` razen če je `endp` enak `NULL`. Če je `base` med `2` in `36`, pretvarjanje predpostavlja, da je vhodno število napisano v tej osnovi. Če je `base` nič, je osnova `8`, `10` ali `16`; vodilna ničla implicira osmiško število, vodilni `0x` ali `0X` implicira šestnajstiško število. Črke v vsakem primeru pomenijo števke od `10` do `base-1`. Če bi odgovor prekoračil obseg, funkcija vrne `LONG_MAX` ali `LONG_MIN`, odvisno od predznaka rezultata, in `errno` se postavi na `ERANGE`.

`unsigned long strtoul(const char *s, char **endp, int base)`

`strtoul` je ista funkcija kot `strtol`, razen da je rezultat tipa `unsigned long` in vrednost v primeru prekoračitve obsega `ULONG_MAX`.

`int rand(void)`

`rand` vrne pseudo-slučajno število v intervalu `0` do `RAND_MAX`, ki je najmanj `32767`.

`void srand(unsigned int seed)`

`srand` zaseje seme `seed` kot nov začetek pseudo-slučajnih števil. Začetno seme je `1`.

`void *calloc(size_t nobj, size_t size)`

`calloc` vrne kazalec na `nobj`-kratni objekt velikost `size` ali `NULL`, če zahtevi ni moč ustreči. Vsi dodeljeni zlogi so inicializirani na nič.

`void *malloc(size_t size)`

`malloc` vrne kazalec na objekt velikost `size` ali `NULL`, če zahtevi ni moč ustreči. Vsi dodeljeni zlogi so neinicializirani.

`void *realloc(void *p, size_t size)`

`realloc` spremeni velikost objekta, na katerega kaže `p`, na `size` zlogov. Vsebina do minimuma stare in nove velikosti ostane nespremenjena. Če je nova velikost večja, so dodatni zlogi neinicializirani. Funkcija `realloc` vrne kazalec na nov prostor ali `NULL`, če zahtevi ni moč ustreči in v tem primeru ostane type*p nespremenjen.

`void free(void *p)`

`free` sprosti prostor, na katerega kaže `p`; če je `p` `NULL`, `free` ne stori nič. Količina `p` mora biti kazalec, ki ga je kdaj prej vrnila funkcija `calloc`, `malloc` ali `realloc`.

`void abort(void)`

`abort` povzroči, da se program konča abnormalno, kot s klicem funkcije `raise(SIGABRT)`.

`void exit(int status)`

`exit` povzroči, da se program konča normalno. Funkcije `atexit` se pokličejo v nasprotnem vrstnem redu, kot so bile registrirane, odprte datoteke izpraznijo izravnalnike, odprti tokovi se zapro in kontrolo prevzame okolje. Kako vrniti status okolju, je določeno z izvedbo, a ničla se smatra za normalen konec. Uporabimo lahko tudi vrednosti `EXIT_SUCCESS` in `EXIT_FAILURE`.

`int atexit(void(*fcn)(void))`

`atexit` registrira funkcijo `fcn`, da jo sistem pokliče ob normalnem izidu programa; funkcija vrne neničelno vrednost, če registracije ni moč opraviti.

`int system(const char *s)`

system preda niz `s` okolju v izvajanje. Če je `s` `NULL`, system vrne neničelno vrednost, če ukazni procesor obstaja. Če je `s` različen od `NULL`, je obnašanje odvisno od izvedbe.

`char *getenv(const char *name)`

`getenv` vrne niz z imenom `name` iz okolja, ali `NULL`, če ne obstaja. Podrobnosti so prepuščene izvedbi.

```
void *bsearch(const void *key, const void *base, size_t n,
    size_t size, int (*cmp)(const void *, const void *))
```

`bsearch` poišče v `base[0], ..., base[n-1]` element, ki se ujema s ključem `*key`. Funkcija `cmp` mora vrniti negativno vrednost, če je njen prvi argument manjši od drugega, nič, če sta enaka in pozitivno vrednost, če je prvi argument večji od drugega. Elementi v večkratni vrednosti `base` morajo biti urejeni po velikosti v naraščajoči vrstni red. Funkcija `bsearch` vrne kazalec na najedeni element in `NULL`, če `bsearch` elementa ne najde.

```
void qsort(void *base, size_t n, size_t size,
    int (*cmp)(const void *, const void *))
```

`qsort` preuredi objekte `base[0], ..., base[n-1]` velikosti `size` zlogov v naraščajoči vrsti red. Primerjalna funkcija `cmp` je taka kot v `bsearch`.

`int abs(int n)`

`abs` vrne absolutno vrednost svojega `int` argumenta.

`long labs(long n)`

`labs` vrne absolutno vrednost svojega `long` argumenta.

`div_t div(int num, int denom)`

`div` izračuna kvocient in ostanek pri deljenju *num/denom*. Rezultati so shranjeni v `int` komponente `quot` in `rem` strukture tipa `div_t`.

`ldiv_t ldiv(long num, long denom)`

`ldiv` izračuna kvocient in ostanek pri deljenju *num/denom*. Rezultati so shranjeni v `long` komponente `quot` in `rem` strukture tipa `ldiv_t`.

C.6 Diagnostika: assert.h

Makro `assert` uporabljamo, da dodamo programu diagnostiko:

```
void assert(int expression)
```

Če je vrednost `expression` ob klicu enaka nič, bo makro `assert` na `stderr` izpisal sporočilo, kot je

```
Assertion failed: expression, file ime, line nnn
```

Makro `assert` potem pokliče `abort`, da konča izvajanje. Izvorno ime in številka vrstice prihajajo iz __FILE__ in __LINE__.

Če je makro `NDEBUG` definiran v trenutku, ko vključimo `<assert.h>`, se makro `assert` obnaša, kot da ga ne bi bilo.

C.7 Variabilne liste argumentov: stdarg.h

Glava `<stdarg.h>` nudi mehanizme za sprehajanje skozi variabilne liste argumentov funkcije z neznanim številom in neznanim tipom argumentov.

Denimo, da je `lastarg` zadnji imenovan parameter funkcije `f` z variabilnim številom argumentov. Tedaj znotraj funkcije `f` deklariramo spremenljivko `ap` tipa `va_list`, ki bo pokazala na vse argumente drugega za drugim:

```
va_list ap;
```

`ap` je treba enkrat inicializirati z makrojem `va_start`, preden posežemo po kakšnem neimenovanem argumentu:

```
va_start(va_list ap, lastarg);
```

V nadaljevanju bo vsako izvajanje makroja `va_arg` vrnilo vrednost, ki ima tip in vrednost naslednjega neimenovanega argumenta in modificiralo `ap` tako, da bo naslednje izvajanje makroja `va_arg` vrnilo naslednji argument:

```
tip va_arg(va_list ap, tip);
```

Makro

```
void va_end(va_list ap);
```

je treba poklicati natanko enkrat potem, ko smo sprocesirali argumente in preden se funkcija konča.

C.8 Nelokalni skoki: setjmp.h

Deklaracije v `<setjmp.h>` nudijo pot, da se izognemo normalnemu zaporedju klica in vrnitve funkcije, tipično z namenom, da omogočimo takojšen povratek iz globoko gnezdenega zaporedja funkcijskih klicev.

```
int setjmp(jmp_buf env)
```

Makro `setjmp` shrani informacijo o stanju v `env`, kjer jo bo uporabila funkcija `longjmp`. Direkten klic `setjmp` vrne nič, neničelno vrednost pa `setjmp` vrne kot rezultat nekega kasnejšega klica `longjmp`. Funkcijo `setjmp` lahko pokličemo brez škode samo v nekaterih kontekstih, v bistvu le v testih `if`, `switch` in v zančnih stavkih ter v enostavnih relacijskih izrazih.

```
    if (setjmp(env) == 0)
        /* get here on direct call */
    else
        /* get here by calling longjmp */
```

`void longjmp(jmp_buf env, int val)`
`longjmp` restavrira stanje, shranjeno v `env` z zadnjim klicem funkcije `setjmp(env)`. Izvajanje se nadaljuje, kot da se je pravkar izvedla funkcija `setjmp` in vrnila neničelno vrednost `val`. Funkcija, ki je izvedla `setjmp`, se med tem ne sme končati. Dostopni objekti imajo vrednosti, kot so jih imeli ob času, ko je bila poklicana funkcija `longjmp`; teh vrednosti `setjmp` ne shrani.

C.9 Signali: signal.h

Glava `<signal.h>` nudi mehanizme za obravnavo izjemnih pogojev, ki se pojavijo med izvajanjem, kot na primer prekinitev iz zunanjega razloga ali napaka med izvajanjem.

`void (*signal(int sig, void (*handler)(int)))(int)`
`signal` odloča, kako bodo obravnavani prihajajoči signali. Če je vrednost kazalca na funkcijo `handler` enaka `SIG_DFL`, se uporabi privzeto obnašanje, ki ga definira izvedba; če je enaka `SIG_IGN`, bo system signal ignoriral; sicer pa bo poklical funkcijo, na katero kaže `handler`, z argumentom, ki identificira signal.
Legalni signali vključujejo

`SIGABRT`	abnormalno terminiranje, na primer iz `abort`
`SIGFPE`	aritmetična napaka, na primer deljenje z nič ali prekoračitev obsega
`SIGILL`	nelegalna slika funkcije, na primer nelegalen ukaz
`SIGINT`	interaktivna zahteva za pozornost, na primer interrupt
`SIGSEGV`	nelegalen dostop do pomnilnika, na primer doseg zunaj meja dodeljenega pomnilnika
`SIGTERM`	zahteva, da se program konča

Funkcija `signal` vrne prejšnjo vrednost kazalca `handler` za specifični signal ali pa `SIG_ERR` v primeru napake.

Ko se v nadaljevanju zgodi signal `sig`, se obravnavalje signala prestavi na privzeto obnašanje in servisna funkcija je poklicana, kot da bi poklicali `(*handler)(sig)`. Če se servisna funkcija vrne, se izvajanje nadaljuje tam, kjer se je signal dogodil.

Začetno stanje signalov je določeno z izvedbo.

`int raise(int sig)`
`raise` pošlje programu signal `sig`; funkcija vrne neničelno vrednost, če ji ne uspe odposlati signala.

C.10 Funkcije za koledar in uro: time.h

Glava `<time.h>` deklarira tipe in funkcije za manipuliranje s koledarjem in uro. Nekatere funkcije procesirajo lokalni čas, ki se lahko loči od koledarskega časa

(`UCT`), ker živimo v različnih časovnih conah. Tipa `clock_t` in `time_t` sta aritmetična tipa, ki predstavljata čase in `struct tm` hrani komponente koledarskega časa:

`int tm_sec;`	sekund po minuti [0, 61] (prestopne sekunde!)
`int tm_min;`	minut po uri [0, 59]
`int tm_hour;`	ur po polnoči [0, 23]
`int tm_mday;`	dan v mesecu [1, 31]
`int tm_mon;`	mesecev po januarju [0, 11]
`int tm_year;`	let po 1900
`int tm_wday;`	dni od nedelje [0, 6]
`int tm_yday;`	dni od 1. januarja [0, 365]
`int tm_isdst;`	zimski čas (Daylight Saving Time)

Podatek `tm_isdst` je poziteven, če je ura premaknjena za eno uro naprej, nič, če ni in negativen, če informacija ni dostopna.

`clock_t clock(void)`

`clock` vrne čas, ki ga je potrošil program od začetka izvajanja, `(clock_t)(-1)`, če podatek ni na voljo. Vrednost `clock()/CLOCKS_PER_SEC` je potrošen čas v sekundah. Ta izjava sugerira, da je ta čas merjen v manjših enotah kot so sekunde.

`time_t time(time_t *tp)`

`time` vrne koledarski čas ali `(time_t)(-1)`, če čas ni znan. Če je `tp` različen od `NULL`, se vrnjena vrednost shrani tudi v `*tp`.

`double difftime(time_t time2, time_t time1)`

`difftime` vrne razliko $time2 - time1$ izraženo v sekundah.

`time_t mktime(struct tm *tp)`

`mktime` pretvori lokalen čas v strukturi `*tp` v koledarski čas v isti reprezentaciji, kot jo uporablja `time`. Komponente morajo biti v navedenih intervalih. Funkcija `mktime` vrne koledarski čas ali `(time_t)(-1)`, če se časa `*tm` ne da predstaviti kot `time_t`.

Naslednje štiri funkcije vrnejo kazalce na statične objekte, ki jih lahko naslednji klici povozijo.

`char *asctime(const struct tm *tp)`

`asctime` pretvori čas v strukturi `*tp` v niz oblike

```
Sun Nov 19 18:13:48 1995
```

in temu nizu sledita še znak za novo vrsto in prazen znak.

`char *ctime(const time_t *tp)`

`ctime` pretvori koledarski čas `*tp` v lokalni čas; klic funkcije je ekvivalenten klicu

```
asctime(localtime(tp))
```

`struct tm *gmtime(const time_t *tp)`

`gmtime` pretvori koledarski čas `*tp` v standardni čas `UTC`. Funkcija vrne `NULL`, če `UTC` ni dostopen. Ime `gmtime` ima zgodovinski pomen (Greenwich Mean Time).

`struct tm *localtime(const time_t *tp)`

`localtime` pretvori koledarski čas `*tp` v lokalni čas.

```
size_t strftime(char *s, size_t smax, const char *fmt,
    const struct tm *tp)
```

`strftime` formatira informacijo o datumu in času iz `*tp` v niz `s` v skladu s formatom `fmt`, ki je analogen `printf` formatu. Običajni znaki (vključno s končnim praznim znakom) se prepišejo v `s`. Vsak `%c` je nadomeščen kot je spodaj opisano z vrednostmi, ki so v skladu z lokalnim okoljem. V `s` se shrani največ `smax` znakov. Funkcija `strftime` vrne število shranjenih znakov brez praznega znaka ali nič, če je potrebnih več kot `smax` znakov.

`%a`	okrajšano ime dneva v tednu
`%A`	polno ime dneva v tednu
`%b`	okrajšano ime meseca
`%B`	polno ime meseca
`%c`	lokalna reprezentacija datuma in časa
`%d`	dan v mesecu (01-31)
`%H`	ura (24 urna ura) (00-23)
`%I`	ura (12 urna ura) (01-12)
`%j`	dan v letu (001-366)
`%m`	mesec (01-12)
`%M`	minuta (00-59)
`%p`	lokalni ekvivalent AM ali PM
`%S`	sekunda (00-59)
`%U`	številka tedna v letu (nedelja je prvi dan tedna) (00-53)
`%w`	dan v tednu (0-6, nedelja je 0)
`%W`	številka tedna v letu (ponedeljek je prvi dan tedna) (00-53)
`%x`	lokalna reprezentacija datuma
`%X`	lokalna reprezentacija časa
`%y`	leto brez stoletja (00-99)
`%Y`	leto s stoletjem
`%Z`	ime časovne zone
`%%`	%

C.11 Omejitve definirane z izvedbo: limits.h in float.h

Glava `<limits.h>` definira konstante za velikosti celoštevilčnih tipov. Vrednosti, ki so spodaj navedene, so minimalne sprejemljive vrednosti; izvedba lahko uporablja večje vrednosti.

`CHAR_BIT`	`8`	bitov v znaku
`CHAR_MAX`	`UCHAR_MAX` ali `SCHAR_MAX`	maksimalna vrednost znaka
`CHAR_MIN`	`0` ali `SCHAR_MIN`	minimalna vrednost znaka
`INT_MAX`	`+32767`	maksimalna vrednost `int`
`INT_MIN`	`-32767`	minimalna vrednost `int`
`LONG_MAX`	`+2147483647L`	maksimalna vrednost long
`LONG_MIN`	`-2147483647L`	minimalna vrednost long
`SCHAR_MAX`	`+127`	maksimalna vrednost predznačenega

		znaka
SCHAR_MIN	-127	minimalna vrednost predznačenega znaka
SHRT_MAX	+32767	maksimalna vrednost short
SHRT_MIN	-32767	minimalna vrednost short
UCHAR_MAX	255U	maksimalna vrednost nepredznačenega znaka
UINT_MAX	65535U	maksimalna vrednost nepredznačenega `int`
ULONG_MAX	4294967295UL	maksimalna vrednost nepredznačenega long
USHRT_MAX	65535U	maksimalna vrednost nepredznačenega short

Imena v spodnji tabeli, podmnožica `<float.h>`, so konstante, povezane z aritmetiko v plavajoči vejici. Kadar je podana vrednost, je to minimalna sprejemljiva absolutna vrednost. Vsaka izvedba definira ustrezne vrednosti.

FLT_RADIX	2	osnova reprezentacije eksponenta, n.pr. `2`, `16`
FLT_ROUNDS		način zaokrožanja pri seštevanju
FLT_DIG	6	desetiških števk natančnosti
FLT_EPSILON	1E-5	najmanjše pozitivno število x, da je $1.0 + x \neq 1.0$
FLT_MANT_DIG		število števk (baza `FLT_RADIX`) v mantisi
FLT_MAX	1E+37	maksimalno število v plavajoči vejici
FLT_MAX_EXP		maksimalen n, da je `FLT_RADIX`$^n - 1$ predstavljivo število
FLT_MIN	1E-37	minimalno normalizirano število v plavajoči vejici
FLT_MIN_EXP		minimalen n, da je `FLT_RADIX`^{n} normalizirano število
DBL_DIG	10	desetiških števk natančnosti
DBL_EPSILON	1E-9	najmanjše pozitivno število x, da je $1.0 + x \neq 1.0$
DBL_MANT_DIG		število števk (baza `FLT_RADIX`) v mantisi
DBL_MAX	1E+37	maksimalno število v plavajoči vejici
DBL_MAX_EXP		maksimalen n, da je `FLT_RADIX`$^n - 1$ predstavljivo število
DBL_MIN	1E-37	minimalno normalizirano število v plavajoči vejici
DBL_MIN_EXP		minimalen n, da je `FLT_RADIX`^{n} normalizirano število

Literatura

[1] Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language.* Prentice Hall Software Series, Englewood Cliffs, New Jersey, 1978.

[2] Brian W. Kernighan, Dennis M. Ritchie. *The C Programming Language, Second Edition.* Prentice Hall Software Series, Englewood Cliffs, New Jersey, 1988.

[3] Herbert Schildt. *The Annotated ANSI C Standard, American National Standard for Programming Languages - C, ANSI/ISO 9899-1990* Osborne McGraw-Hill, Berkeley, California, 1990.

[4] Samuel P. Harbison, Guy L. Steele Jr. *C, A Reference Manual, Third Edition.* Prentice Hall Software Series, Englewood Cliffs, New Jersey, 1991.

[5] P. J. Plauger. *The Standard C Library.* Prentice Hall Software Series, Englewood Cliffs, New Jersey, 1992.

[6] *Purify, User's Guide,* Pure Software, Sunnyvale, California, 1992.

Stvarno kazalo

argc - argument count, 110
argv - argument vector, 110
ASCII, 14
asociativnost, 51
asociativnost operatorjev, 59
assembler, 18
avtomatičen pomnilniški razred, 77

bel znak, 17, 20
bingljajoči else, 64
bitno polje, 132
bločno strukturiran jezik, 83
blok, 63, 83
byte, 13

C, 13
    ANSI, 13
    K&R, 13
C++, 13
celoštevilčna vrednost, 20

datoteka, 31, 151
definicija
    spremenljivk, 48
definicija ali deklaracija, 81
deklaracija ali definicija, 81
dodeljevanje, 49
doseg imena, 80

eksteren objekt, 77
else
    bingljajoči, 64
enumeracije, 48

formatni niz, 22
Fortran, 13
funkcija
    acos, 150
    addtree, 126
    afree, 103
    alloc, 103
    asin, 149
    atan, 150
    atan2, 150
    atof, 75
    atoi, 30, 56
    binsearch, 65
    bitcount, 55
    calloc, 149
    ceil, 150
    clearerr, 146
    close, 154
    cos, 149
    cosh, 150
    day_of_year, 108
    dcl, 115
    defined, 90
    dirdcl, 115
    dmatrix, 113
    exit, 146
    exp, 149
    fabs, 150
    fclose, 144
    feof, 146
    ferror, 146
    fgets, 146
    floor, 150
    fmod, 32
    fopen, 144
    fprintf, 139, 145
    fputs, 146
    free, 103, 149
    fscanf, 145
    fseek, 155
    getbits, 54
    getc, 145
    getch, 80
    getchar, 33, 135, 145, 153
    getint, 96
    hash, 127
    install, 126
    isalnum, 57
    isalpha, 57
    isascii, 57
    iscntrl, 57
    isdigit, 37, 57
    isgraph, 57
    islower, 57
    isprint, 57

  - ispunct, 57
  - isspace, 37, 57
  - isupper, 57
  - isxdigit, 57
  - itoa, 68
  - log, 149
  - log10, 149
  - lokalne spremenljivke, 40
  - lookup, 126
  - lower, 57
  - lseek, 155
  - main, 15
  - makefraction, 119
  - malloc, 49, 103, 149
  - minprintf, 140
  - month_day, 108
  - month_name, 109
  - myatof, 75
  - myatoi, 56, 68, 76
  - mygetchar, 153
  - myqsort, 112
  - mystrcat, 52
  - mystrcmp, 106
  - mystrcpy, 105
  - mystrlen, 47, 101, 103
  - open, 153
  - perror, 153
  - pow, 39, 149
  - power, 39
  - prenos parametrov, 41
  - printd, 85
  - printf, 15, 136, 137
  - prototip, 40
  - putc, 145
  - putchar, 33, 136, 145
  - qsort, 107
  - rand, 59
  - read, 152
  - realloc, 149
  - remove, 154
  - sbrk, 157
  - scanf, 141
  - shellsort, 107
  - sin, 58, 149
  - sinf, 58
  - sinh, 150
  - sprintf, 139
  - sqrt, 149
  - squeeze, 52
  - srand, 59
  - statična, 40
  - strcat, 52, 147
  - strchr, 148
  - strcmp, 148
  - strcpy, 105, 148
  - strcspn, 148
  - strdup, 126, 148
  - strindex, 74
  - strlen, 47, 148
  - strncat, 147
  - strncmp, 148
  - strncpy, 148
  - strpbrk, 148
  - strrchr, 148
  - strspn, 148
  - strstr, 74, 111, 148
  - strtok, 148
  - swap, 95
  - system, 148
  - tan, 149
  - tanh, 150
  - tolower, 57, 137
  - toupper, 57
  - trim, 69
  - ungetc, 80, 146
  - ungetch, 80
  - vfprintf, 140
  - vprintf, 140
  - vsprintf, 140
  - write, 152
- funkcije, 39

- glava, 15
  - ctype.h, 137
  - math.h, 149
  - stdarg.h, 140
  - stdio.h, 136
  - stdlib.h, 148
  - string.h, 147
  - unistd.h, 152
- globalna spremenljivka, 78

- IEEE standard, 20
- imenik, 151
- indeks, 23
- infiksna notacija, 78
- inicializacija, 84
- inicializacija spremenljivk, 23
- interen objekt, 77
- interpretacija
  - nepredznačena, 13
  - predznačena, 13
  - spreminjanje, 14
- ISO, 14
- ISO 8859, 14
- izravnalnik, 153
- izraz
  - konstanten, 46

    pogojni, 27, 56

kategorija, 77
    auto, 77
    extern, 77
    register, 77
    static, 77
    typedef, 77, 128
kazalci
    in večkratna vrednost, 93
kazalec, 93
    kot argument funkcije, 95
    na strukturo, 120
    na znak, 29
knjižnica
    libc.a, 18
    libm.a, 32
komentar, 20
končno stanje, 15
konstanta, 45
    enumeracijska, 48
    int, 45
    long, 45
    niz, 47
    unsigned int, 45
    unsigned long, 45
konstanten niz
    stik, 47
kvalifikacija
    const, 50

Latin 2 Code Set, 14
LIFO
    Last In First Out, 103
lupina, 15

makro, 86
    brez parametrov, 34
    EOF, 16, 34
makro procesor, 18

naslov, 93
nova vrsta, 17

objekt
    eksteren, 77
    interen, 77
obratna Poljska notacija, 78
operator
    adresni, 93
    bitni ekskluzivni ALI, 53
    bitni eniški komplement, 53
    bitni IN, 53
    bitni inkluzivni ALI, 53
    bitni pomik v desno, 53
    bitni pomik v levo, 53
    citiranja, 88
    deljenja, 22, 50
    dereferenciranja, 94
    enak, 22, 51
    logična negacija, 51
    logični ALI, 37, 51, 53
    logični IN, 37, 51, 53
    manjši, 22, 51
    manjši ali enak, 22, 51
    množenja, 50
    neenak, 22, 51
    odštevanja, 50
    ostanka, 22, 50
    povečevanja, 26, 52
    prisile, 58
    s prirejanjem, 26, 54
    seštevanja, 50
    sizeof, 21, 24
    stikanja, 89
    večji, 22, 51
    večji ali enak, 22, 51
    vejica, 68
    zmanjševanja, 26, 52
optimizator, 18

Pascal, 13
pomnilniški razred
    avtomatičen, 77
    statičen, 77
povezovalnik, 18
predelčnik, 17
predprocesor, 18, 86
    direktiva define, 86
    direktiva elif, 89
    direktiva else, 89
    direktiva endif, 89
    direktiva if, 89
    direktiva ifdef, 90
    direktiva ifndef, 90
    direktiva include, 86
    direktiva undef, 88
predprocesorjeve direktive, 18
predznačenost tipa char, 14
prelomnica, 157
presledek, 17
prioritete, 51
prioritete operatorjev, 59
prirejanje, 21
prisila, 34
program
    binsearch, 65
    calc, 78, 82
    count, 37

  count2, 66
  dayofyear, 108
  dcl, 114
  fib, 85
  fract, 120
  freq, 126
  grep, 73
  hello, 15
  itoa, 68
  keyword1, 123
  keyword2, 123
  lint, 75
  longest, 42
  lower, 136
  macro, 128
  matrix, 114
  minprintf, 140
  money, 19
  money2, 24
  money3, 27
  money4, 27
  money5, 30
  money6, 32
  myatoi, 68
  mycat, 145
  mycat2, 146
  mycp, 152
  mycp1, 154
  myecho, 30
  mygetchar, 153
  mygetchar2, 153
  mygrep, 73, 111
  mygrep1, 111
  mygrep2, 111
  mywc, 37
  quicksort, 85
  reverse, 68
  shellsort, 68
  sortlines, 107
  sortlines2, 112
  xcalc1, 142

računalnik
  model, 13
  pomnilnik, 13
razpršena tabela, 126
razširjeni ASCII kod, 14
realna števila, 46
register spremenljivke, 83
rekurzija, 85
Ritchie, Dennis, 13

SCCS, 19
sestavljeni stavek, 15
shell, 15
sklad, 103
spremenljivka, 20
  avtomatična, 49, 83
  globalna, 78
  statična, 49
  statična interna, 83
  začetna vrednost, 49, 84
standardni deskriptor, 152
  STDERR_FILENO, 152
  STDIN_FILENO, 152
  STDOUT_FILENO, 152
statičen pomnilniški razred, 77
stavek
  break, 66, 69
  continue, 69
  do, 25, 68
  for, 25, 67
  goto, 69
  if, 27
  pogojni, 63
  return, 40, 74
  sestavljeni, 15, 25, 63, 83
  switch, 65
  while, 24, 66
storage class, 77
stranski učinki, 60
Structran, 13
struktura, 117
  inicializacija, 118
  legalne operacije, 119
  samoreferencirajoča, 125
  večkratna, 122
širina, 14
štrcelj, 74

termination status, 15
tip, 21
  char, 21, 46
  double, 21, 46
  float, 20, 21, 46
  int, 20, 21
  long, 21, 36
  long double, 46
  short, 21
  size_t, 47, 129
  ssize_t, 152

ubežna zaporedja, 46
unija, 129
  inicializacija, 132
UNIX, 13

variabilne liste argumentov, 139

večkratna vrednost, 23
    in kazalci, 93
vrednost
    prirejanja, 35
    relacije, 51

zanka
    neskončna, 24
zapis
    desetiški, 45
    osmiški, 45
    šestnajstiški, 45
zbirni jezik, 18
zbirnik, 18
zlog, 13
značka, 117
znak, 14
    prazen, 47
znaki, 46
znakovni kod, 14